Arquitetura de Computadores:

Uma Abordagem Quantitativa

Quinta Edição

David A. Patterson
John L. Hennessy

ARQUITETURA DE COMPUTADORES

UMA ABORDAGEM QUANTITATIVA

Tradução da 5ª Edição

Tradução: Eduardo Kraszczuk

Revisão Técnica: Ricardo Pannain

Do original: *Computer Architecture: A Quantitative Approach*
Tradução autorizada do idioma inglês da edição publicada por Morgan Kaufmann, an imprint of Elsevier, Inc.

Copidesque: Andréa Vidal
Revisão Gráfica: Adriana Maria Patrício Takaki / Marco Antonio Corrêa / Roberto Mauro dos Santos Facce:
Editoração Eletrônica: Thomson Digital

Elsevier Editora Ltda.
Conhecimento sem Fronteiras
Rua Sete de Setembro, 111 – 16º andar
20050-006 – Centro - Rio de Janeiro – RJ - Brasil

Rua Quintana, 753/8º andar
04569-011 Brooklin - São Paulo - SP - Brasil

Serviço de Atendimento ao Cliente
O800-0265340
Atendimento1@elsevier.com

ISBN: 978-85-352-6122-6
ISBN (versão digital): 978-85-352-6411-1
Edição original: ISBN 978-0-12-383872-8

Nota: Muito zelo e técnica foram empregados na edição desta obra. No entanto, podem ocorrer erros de digitação, impressão ou dúvida conceitual. Em qualquer das hipóteses, solicitamos a comunicação ao nosso Serviço de Atendimento ao Cliente, para que possamos esclarecer ou encaminhar a questão.

Nem a editora nem o autor assumem qualquer responsabilidade por eventuais danos ou perdas a pessoas ou bens, ou bens, originados do uso desta publicação.

CIP-BRASIL. CATALOGAÇÃO NA PUBLICAÇÃO
SINDICATO NACIONAL DOS EDITORES DE LIVROS, RJ

H436a
Hennessy, John L.
Arquitetura de computadores : uma abordagem quantitativa / John L. Hennessy, David A. Petterson ; tradução Eduardo Kraszczuk. - [5. ed.] - Rio de Janeiro : Elsevier, 2014.
744 p. : il. ; 28 cm.

Tradução de: Computer architecture, 5th ed. : a quantitative approach
Inclui apêndice
ISBN 978-85-352-6122-6

1. Arquitetura de computador. I. Patterson, David A. II. Título.

13-05666 CDD: 004.22
CDU: 004.2

Sobre os Autores

John L. Hennessy é o décimo presidente da Universidade de Stanford, onde é membro do corpo docente desde 1977, nos departamentos de Engenharia Elétrica e Ciência da Computação. Hennessy é membro do IEEE e ACM, membro da Academia Nacional de Engenharia e da Sociedade Americana de Filosofia e membro da Academia Americana de Artes e Ciências. Entre seus muitos prêmios estão o Prêmio Eckert-Mauchly de 2001, por suas contribuições para a tecnologia RISC, o Prêmio Seymour Cray de Engenharia da Computação de 2001 e o Prêmio John von Neumann de 2000, que ele dividiu com David Patterson. Ele também recebeu sete doutorados honorários.

Em 1981, ele iniciou o Projeto MIPS, em Stanford, com um grupo de estudantes de pós-graduação. Depois de completar o projeto em 1984, tirou licença da universidade para co-fundar a MIPS Computer Systems (hoje MIPS Technologies), que desenvolveu um dos primeiros microprocessadores RISC comerciais. Em 2006, mais de 2 bilhões de microprocessadores MIPS foram vendidos em dispositivos, variando de video games e computadores palmtop a impressoras laser e switches de rede. Em seguida, Hennessy liderou o projeto DASH (Director Architeture for Shared Memory – Arquitetura Diretora para Memória Compartilhada), que criou o protótipo do primeiro microprocessador com cache coerente escalável. Muitas das ideias-chave desse projeto foram adotadas em multiprocessadores modernos. Além de suas atividades técnicas e responsabilidades na universidade, ele continuou a trabalhar com diversas empresas startup como conselheiro nos estágios iniciais e como investidor.

David A. Patterson ensina arquitetura de computadores na Universidade da Califórnia, em Berkeley, desde que se juntou ao corpo docente em 1977, onde ele ocupa a Cadeira Pardee de Ciência da Computação. Sua docência foi honrada com o Prêmio de Ensino Notável da Universidade da Califórnia, o Prêmio Karlstrom da ACM, a Medalha Mulligan de Educação e o Prêmio de Ensino Universitário do IEEE. Patterson recebeu o Prêmio de Realização Técnica do IEEE e o Prêmio Eckert-Mauchly por contribuições para o RISC e dividiu o Prêmio Johnson de Armazenamento de Informações por contribuições para o RAID. Ele também dividiu a Medalha John von Neumann do IEEE e o Prêmio C&C com John Hennessy. Como seu coautor, Patterson é membro da Academia Americana de Artes e Ciências, do Museu da História dos Computadores, ACM e IEEE, e foi eleito para a Academia Nacional de Engenharia, Academia Nacional de Ciências e para o Hall da Fama da Engenharia do Vale do Silício. Ele atuou no Comitê Consultivo de Tecnologia da Informação do presidente dos Estados Unidos, como presidente da divisão de CS no departamento EECS em Berkeley, como presidente da Associação de Pesquisa em Computação e como Presidente da ACM. Este histórico levou a prêmios de Serviço Destacado da ACM e CRA.

Em Berkeley, Patterson liderou o projeto e a implementação do RISC I, provavelmente o primeiro computador com conjunto reduzido de instruções VLSI, e a fundação da arquitetura comercial SPARC. Ele foi líder do projeto Arrays Redundantes de Discos Baratos (Redundant Array of Inexpensive Disks – RAID), que levou a sistemas de armazenamento

confiáveis para muitas empresas. Ele também se envolveu no projeto Rede de Workstations (Network of Workstations – NOW), que levou à tecnologia de clusters usada pelas empresas de Internet e, mais tarde, à computação em nuvem. Esses projetos valeram três prêmios de dissertação da ACM. Seus projetos de pesquisa atuais são o Laboratório Algoritmo-Máquina-Pessoas e o Laboratório de Computação Paralela, onde ele é o diretor. O objetivo do Laboratório AMP é desenvolver algoritmos de aprendizado de máquina escaláveis, modelos de programação amigáveis para computadores em escala de depósito e ferramentas de crowd-sourcing para obter rapidamente insights valiosos de muitos dados na nuvem. O objetivo do laboratório Par é desenvolver tecnologias para entregar softwares escaláveis, portáveis, eficientes e produtivos para dispositivos pessoais móveis paralelos.

Para Andrea, Linda, e nossos quatro filhos

Elogios para *Arquitetura de Computadores: Uma Abordagem Quantitativa* Quinta Edição

"A 5ª edição de *Arquitetura de Computadores: Uma Abordagem Quantitativa* continua o legado, fornecendo aos estudantes de arquitetura de computadores as informações mais atualizadas sobre as plataformas computacionais atuais e insights arquitetônicos para ajudá-los a projetar sistemas futuros. Um destaque da nova edição é o capítulo significativamente revisado sobre paralelismo em nível de dados, que desmistifica as arquiteturas de GPU com explicações claras, usando terminologia tradicional de arquitetura de computadores."

—Krste Asanovic, Universidade da Califórnia, Berkeley

"*Arquitetura de Computadores: Uma Abordagem Quantitativa* é um clássico que, como um bom vinho, fica cada vez melhor. Eu comprei meu primeiro exemplar quando estava terminando a graduação e ele continua sendo um dos volumes que eu consulto com mais frequência. Quando a quarta edição saiu, havia tanto conteúdo novo que eu precisava comprá-la para continuar atualizado. E, enquanto eu revisava a quinta edição, percebi que Hennessy e Patterson tiveram sucesso de novo. Todo o conteúdo foi bastante atualizado e só o Capítulo 6 já torna esta nova edição uma leitura necessária para aqueles que realmente querem entender a computação em nuvem e em escala de depósito. Somente Hennessy e Patterson têm acesso ao pessoal do Google, Amazon, Microsoft e outros provedores de computação em nuvem e de aplicações em escala de Internet, e não existe melhor cobertura dessa importante área em outro lugar da indústria."

—James Hamilton, Amazon Web Services

"Hennessy e Patterson escreveram a primeira edição deste livro quando os estudantes de pós-graduação construíam computadores com 50.000 transistores. Hoje, computadores em escala de depósito contêm esse mesmo número de servidores, cada qual consistindo de dúzias de processadores independentes e bilhões de transistores. A evolução da arquitetura de computadores tem sido rápida e incansável, mas *Arquitetura de Computadores: Uma Abordagem Quantitativa* acompanhou o processo com cada edição explicando e analisando com precisão as importantes novas ideias que tornam esse campo tão excitante."

—James Larus, Microsoft Research

"Esta nova adição adiciona um soberbo novo capítulo sobre paralelismo em nível de dados em SIMD de vetor e arquiteturas de GPU. Ele explica conceitos-chave de arquitetura no interior das GPUs de mercado de massa, mapeando-os para termos tradicionais e comparando-os com arquiteturas de vetor e SIMD. Ele chega no momento certo e é relevante à mudança generalizada para a computação por GPU paralela. *Arquitetura de Computadores: Uma Abordagem Quantitativa* continua sendo o primeiro a apresentar uma cobertura completa da arquitetura de importantes novos desenvolvimentos!"

—John Nickolls, NVIDIA

"A nova edição deste livro – hoje um clássico – destaca a ascendência do paralelismo explícito (dados, thread, requisição) dedicando um capítulo inteiro a cada tipo. O capítulo sobre paralelismo de dados é particularmente esclarecedor: a comparação e o contraste

entre SIMD de vetor, SIMD em nível de instrução e GPU ultrapassam o jargão associado a cada arquitetura e expõem as similaridades e diferenças entre elas."

—Kunle Olukotun, Universidade de Stanford

"A 5ª edição de *Arquitetura de Computadores: Uma Abordagem Quantitativa* explora os diversos conceitos paralelos e seus respectivos *trade-offs*. Assim como as edições anteriores, esta nova edição cobre as mais recentes tendências tecnológicas. Um destaque é o grande crescimento dos dispositivos pessoais móveis (Personal Mobile Devices – PMD) e da computação em escala de depósito (Warehouse-Scale Computing – WSC), cujo foco mudou para um equilíbrio mais sofisticado entre desempenho e eficiência energética em comparação com o desempenho bruto. Essas tendências estão alimentando nossa demanda por mais capacidade de processamento, que, por sua vez, está nos levando mais longe no caminho paralelo."

—Andrew N. Sloss, Engenheiro consultor, ARM
Autor de *ARM System Developer's Guide*

Agradecimentos

Embora este livro ainda esteja na quinta edição, criamos dez versões diferentes do conteúdo: três versões da primeira edição (alfa, beta e final) e duas versões da segunda, da terceira e da quarta edições (beta e final). Nesse percurso, recebemos a ajuda de centenas de revisores e usuários. Cada um deles ajudou a tornar este livro melhor. Por isso, decidimos fazer uma lista de todas as pessoas que colaboraram em alguma versão deste livro.

COLABORADORES DA QUINTA EDIÇÃO

Assim como nas edições anteriores, este é um esforço comunitário que envolve diversos voluntários. Sem a ajuda deles, esta edição não estaria tão bem acabada.

Revisores

Jason D. Bakos, University of South Carolina; Diana Franklin, The University of California, Santa Barbara; Norman P. Jouppi, HP Labs; Gregory Peterson, University of Tennessee; Parthasarathy Ranganathan, HP Labs; Mark Smotherman, Clemson University; Gurindar Sohi, University of Wisconsin–Madison; Mateo Valero, Universidad Politécnica de Cataluña; Sotirios G. Ziavras, New Jersey Institute of Technology.

Membros do Laboratório Par e Laboratório RAD da University of California–Berkeley, que fizeram frequentes revisões dos Capítulos 1, 4 e 6, moldando a explicação sobre GPUs e WSCs: Krste Asanovic, Michael Armbrust, Scott Beamer, Sarah Bird, Bryan Catanzaro, Jike Chong, Henry Cook, Derrick Coetzee, Randy Katz, Yun-sup Lee, Leo Meyervich, Mark Murphy, Zhangxi Tan, Vasily Volkov e Andrew Waterman.

Painel consultivo

Luiz André Barroso, Google Inc.; Robert P. Colwell, R&E Colwell & Assoc. Inc.; Krisztian Flautner, VP de R&D na ARM Ltd.; Mary Jane Irwin, Penn State; David Kirk, NVIDIA; Grant Martin, cientista-chefe, Tensilica; Gurindar Sohi, University of Wisconsin–Madison; Mateo Valero, Universidad Politécnica de Cataluña.

Apêndices

Krste Asanovic, University of California–Berkeley (Apêndice G); Thomas M. Conte, North Carolina State University (Apêndice E); José Duato, Universitat Politècnica de València and Simula (Apêndice F); David Goldberg, Xerox PARC (Apêndice J); Timothy M. Pinkston, University of Southern California (Apêndice F).

José Flich, da Universidad Politécnica de Valencia, deu contribuições significativas para a atualização do Apêndice F.

Estudos de caso e exercícios

Jason D. Bakos, University of South Carolina (Capítulos 3 e 4); Diana Franklin, University of California, Santa Barbara (Capítulo 1 e Apêndice C); Norman P. Jouppi, HP Labs (Capítulo 2); Naveen Muralimanohar, HP Labs (Capítulo 2); Gregory Peterson, University of Tennessee (Apêndice A); Parthasarathy Ranganathan, HP Labs (Capítulo 6); Amr Zaky, University of Santa Clara (Capítulo 5 e Apêndice B).

Jichuan Chang, Kevin Lim e Justin Meza auxiliaram no desenvolvimento de testes dos estudos de caso e exercícios do Capítulo 6.

Material adicional

John Nickolls, Steve Keckler e Michael Toksvig da NVIDIA (Capítulo 4, NVIDIA GPUs); Victor Lee, Intel (Capítulo 4, comparação do Core i7 e GPU); John Shalf, LBNL (Capítulo 4, arquiteturas recentes de vetor); Sam Williams, LBNL (modelo *roofline* para computadores no Capítulo 4); Steve Blackburn, da Australian National University, e Kathryn McKinley, da University of Texas, em Austin (Desempenho e medições de energia da Intel, no Capítulo 5); Luiz Barroso, Urs Hölzle, Jimmy Clidaris, Bob Felderman e Chris Johnson do Google (Google WSC, no Capítulo 6); James Hamilton, da Amazon Web Services (Distribuição de energia e modelo de custos, no Capítulo 6).

Jason D. Bakos. da University of South Carolina, desenvolveu os novos *slides* de aula para esta edição.

Mais uma vez, nosso agradecimento especial a Mark Smotherman, da Clemson University, que fez a leitura técnica final do nosso manuscrito. Mark encontrou diversos erros e ambiguidades, e, em consequência disso, o livro ficou muito mais limpo.

Este livro não poderia ter sido publicado sem uma editora, é claro. Queremos agradecer a toda a equipe da Morgan Kaufmann/Elsevier por seus esforços e suporte. Pelo trabalho nesta edição, particularmente, queremos agradecer aos nossos editores Nate McFadden e Todd Green, que coordenaram o painel consultivo, o desenvolvimento dos estudos de caso e exercícios, os grupos de foco, as revisões dos manuscritos e a atualização dos apêndices.

Também temos de agradecer à nossa equipe na universidade, Margaret Rowland e Roxana Infante, pelas inúmeras correspondências enviadas e pela "guarda do forte" em Stanford e Berkeley enquanto trabalhávamos no livro.

Nosso agradecimento final vai para nossas esposas, pelo sofrimento causado pelas leituras, trocas de ideias e escrita realizadas cada vez mais cedo todos os dias.

COLABORADORES DAS EDIÇÕES ANTERIORES

Revisores

George Adams, Purdue University; Sarita Adve, University of Illinois, Urbana–Champaign; Jim Archibald, Brigham Young University; Krste Asanovic, Massachusetts Institute of Technology; Jean-Loup Baer, University of Washington; Paul Barr, Northeastern University; Rajendra V. Boppana, University of Texas, San Antonio; Mark Brehob, University of Michigan; Doug Burger, University of Texas, Austin; John Burger, SGI; Michael Butler; Thomas Casavant; Rohit Chandra; Peter Chen, University of Michigan; as turmas de SUNY Stony Brook, Carnegie Mellon, Stanford, Clemson e Wisconsin; Tim Coe, Vitesse Semiconductor; Robert P. Colwell; David Cummings; Bill Dally; David Douglas; José Duato, Universitat Politècnica de València and Simula; Anthony Duben, Southeast Missouri State University; Susan Eggers, University of Washington; Joel Emer; Barry Fagin, Dartmouth; Joel Ferguson, University of California, Santa

Cruz; Carl Feynman; David Filo; Josh Fisher, Hewlett-Packard Laboratories; Rob Fowler, DIKU; Mark Franklin, Washington University (St. Louis); Kourosh Gharachorloo; Nikolas Gloy, Harvard University; David Goldberg, Xerox Palo Alto Research Center; Antonio González, Intel and Universitat Politècnica de Catalunya; James Goodman, University of Wisconsin–Madison; Sudhanva Gurumurthi, University of Virginia; David Harris, Harvey Mudd College; John Heinlein; Mark Heinrich, Stanford; Daniel Helman, University of California, Santa Cruz; Mark D. Hill, University of Wisconsin–Madison; Martin Hopkins, IBM; Jerry Huck, Hewlett-Packard Laboratories; Wen-mei Hwu, University of Illinois at Urbana–Champaign; Mary Jane Irwin, Pennsylvania State University; Truman Joe; Norm Jouppi; David Kaeli, Northeastern University; Roger Kieckhafer, University of Nebraska; Lev G. Kirischian, Ryerson University; Earl Killian; Allan Knies, Purdue University; Don Knuth; Jeff Kuskin, Stanford; James R. Larus, Microsoft Research; Corinna Lee, University of Toronto; Hank Levy; Kai Li, Princeton University; Lori Liebrock, University of Alaska, Fairbanks; Mikko Lipasti, University of Wisconsin–Madison; Gyula A. Mago, University of North Carolina, Chapel Hill; Bryan Martin; Norman Matloff; David Meyer; William Michalson, Worcester Polytechnic Institute; James Mooney; Trevor Mudge, University of Michigan; Ramadass Nagarajan, University of Texas at Austin; David Nagle, Carnegie Mellon University; Todd Narter; Victor Nelson; Vojin Oklobdzija, University of California, Berkeley; Kunle Olukotun, Stanford University; Bob Owens, Pennsylvania State University; Greg Papadapoulous, Sun Microsystems; Joseph Pfeiffer; Keshav Pingali, Cornell University; Timothy M. Pinkston, University of Southern California; Bruno Preiss, University of Waterloo; Steven Przybylski; Jim Quinlan; Andras Radics; Kishore Ramachandran, Georgia Institute of Technology; Joseph Rameh, University of Texas, Austin; Anthony Reeves, Cornell University; Richard Reid, Michigan State University; Steve Reinhardt, University of Michigan; David Rennels, University of California, Los Angeles; Arnold L. Rosenberg, University of Massachusetts, Amherst; Kaushik Roy, Purdue University; Emilio Salgueiro, Unysis; Karthikeyan Sankaralingam, University of Texas at Austin; Peter Schnorf; Margo Seltzer; Behrooz Shirazi, Southern Methodist University; Daniel Siewiorek, Carnegie Mellon University; J. P. Singh, Princeton; Ashok Singhal; Jim Smith, University of Wisconsin–Madison; Mike Smith, Harvard University; Mark Smotherman, Clemson University; Gurindar Sohi, University of Wisconsin–Madison; Arun Somani, University of Washington; Gene Tagliarin, Clemson University; Shyamkumar Thoziyoor, University of Notre Dame; Evan Tick, University of Oregon; Akhilesh Tyagi, University of North Carolina, Chapel Hill; Dan Upton, University of Virginia; Mateo Valero, Universidad Politécnica de Cataluña, Barcelona; Anujan Varma, University of California, Santa Cruz; Thorsten von Eicken, Cornell University; Hank Walker, Texas A&M; Roy Want, Xerox Palo Alto Research Center; David Weaver, Sun Microsystems; Shlomo Weiss, Tel Aviv University; David Wells; Mike Westall, Clemson University; Maurice Wilkes; Eric Williams; Thomas Willis, Purdue University; Malcolm Wing; Larry Wittie, SUNY Stony Brook; Ellen Witte Zegura, Georgia Institute of Technology; Sotirios G. Ziavras, New Jersey Institute of Technology.

Apêndices

O apêndice sobre vetores foi revisado por Krste Asanovic, do Massachusetts Institute of Technology. O apêndice sobre ponto flutuante foi escrito originalmente por David Goldberg, da Xerox PARC.

Exercícios

George Adams, Purdue University; Todd M. Bezenek, University of Wisconsin–Madison (em memória de sua avó, Ethel Eshom); Susan Eggers; Anoop Gupta; David Hayes; Mark Hill; Allan Knies; Ethan L. Miller, University of California, Santa Cruz; Parthasarathy Ranganathan, Compaq Western Research Laboratory; Brandon Schwartz, University of

Wisconsin–Madison; Michael Scott; Dan Siewiorek; Mike Smith; Mark Smotherman; Evan Tick; Thomas Willis

Estudos de caso e exercícios

Andrea C. Arpaci-Dusseau, University of Wisconsin–Madison; Remzi H. Arpaci Dusseau, University of Wisconsin–Madison; Robert P. Colwell, R&E Colwell & Assoc., Inc.; Diana Franklin, California Polytechnic State University, San Luis Obispo; Wen-mei W. Hwu, University of Illinois em Urbana–Champaign; Norman P. Jouppi, HP Labs; John W. Sias, University of Illinois em Urbana–Champaign; David A. Wood, University of Wisconsin–Madison

Agradecimentos especiais

Duane Adams, Defense Advanced Research Projects Agency; Tom Adams; Sarita Adve, University of Illinois, Urbana–Champaign; Anant Agarwal; Dave Albonesi, University of Rochester; Mitch Alsup; Howard Alt; Dave Anderson; Peter Ashenden; David Bailey; Bill Bandy, Defense Advanced Research Projects Agency; Luiz Barroso, Compaq's Western Research Lab; Andy Bechtolsheim; C. Gordon Bell; Fred Berkowitz; John Best, IBM; Dileep Bhandarkar; Jeff Bier, BDTI; Mark Birman; David Black; David Boggs; Jim Brady; Forrest Brewer; Aaron Brown, University of California, Berkeley; E. Bugnion, Compaq's Western Research Lab; Alper Buyuktosunoglu, University of Rochester; Mark Callaghan; Jason F. Cantin; Paul Carrick; Chen-Chung Chang; Lei Chen, University of Rochester; Pete Chen; Nhan Chu; Doug Clark, Princeton University; Bob Cmelik; John Crawford; Zarka Cvetanovic; Mike Dahlin, University of Texas, Austin; Merrick Darley; the staff of the DEC Western Research Laboratory; John DeRosa; Lloyd Dickman; J. Ding; Susan Eggers, University of Washington; Wael El-Essawy, University of Rochester; Patty Enriquez, Mills; Milos Ercegovac; Robert Garner; K. Gharachorloo, Compaq's Western Research Lab; Garth Gibson; Ronald Greenberg; Ben Hao; John Henning, Compaq; Mark Hill, University of Wisconsin–Madison; Danny Hillis; David Hodges; Urs Hölzle, Google; David Hough; Ed Hudson; Chris Hughes, University of Illinois em Urbana–Champaign; Mark Johnson; Lewis Jordan; Norm Jouppi; William Kahan; Randy Katz; Ed Kelly; Richard Kessler; Les Kohn; John Kowaleski, Compaq Computer Corp; Dan Lambright; Gary Lauterbach, Sun Microsystems; Corinna Lee; Ruby Lee; Don Lewine; Chao-Huang Lin; Paul Losleben, Defense Advanced Research Projects Agency; Yung-Hsiang Lu; Bob Lucas, Defense Advanced Research Projects Agency; Ken Lutz; Alan Mainwaring, Intel Berkeley Research Labs; Al Marston; Rich Martin, Rutgers; John Mashey; Luke McDowell; Sebastian Mirolo, Trimedia Corporation; Ravi Murthy; Biswadeep Nag; Lisa Noordergraaf, Sun Microsystems; Bob Parker, Defense Advanced Research Projects Agency; Vern Paxson, Center for Internet Research; Lawrence Prince; Steven Przybylski; Mark Pullen, Defense Advanced Research Projects Agency; Chris Rowen; Margaret Rowland; Greg Semeraro, University of Rochester; Bill Shannon; Behrooz Shirazi; Robert Shomler; Jim Slager; Mark Smotherman, Clemson University; o SMT research group, University of Washington; Steve Squires, Defense Advanced Research Projects Agency; Ajay Sreekanth; Darren Staples; Charles Stapper; Jorge Stolfi; Peter Stoll; os estudantes de Stanford e de Berkeley, que deram suporte às nossas primeiras tentativas de escrever este livro; Bob Supnik; Steve Swanson; Paul Taysom; Shreekant Thakkar; Alexander Thomasian, New Jersey Institute of Technology; John Toole, Defense Advanced Research Projects Agency; Kees A. Vissers, Trimedia Corporation; Willa Walker; David Weaver; Ric Wheeler, EMC; Maurice Wilkes; Richard Zimmerman.

John Hennessy, David Patterson

Introdução

Por Luiz André Barroso, Google Inc.

A primeira edição de *Arquitetura de Computadores: Uma Abordagem Quantitativa*, de Hennessy e Patterson, foi lançada durante meu primeiro ano na universidade. Eu pertenço, portanto, àquela primeira leva de profissionais que aprenderam a disciplina usando este livro como guia. Sendo a perspectiva um ingrediente fundamental para um prefácio útil, eu me encontro em desvantagem, dado o quanto dos meus próprios pontos de vista foram coloridos pelas quatro edições anteriores deste livro. Outro obstáculo para uma perspectiva clara é que a reverência de estudante a esses dois superastros da Ciência da Computação ainda não me abandonou, apesar de (ou talvez por causa de) eu ter tido a chance de conhecê-los nos anos seguintes. Essas desvantagens são mitigadas pelo fato de eu ter exercido essa profissão continuamente desde a primeira edição deste livro, o que me deu a chance de desfrutar sua evolução e relevância duradora.

A última edição veio apenas dois anos depois que a feroz corrida industrial por maior frequência de clock de CPU chegou oficialmente ao fim, com a Intel cancelando o desenvolvimento de seus núcleos únicos de 4 GHz e abraçando as CPUs multicore. Dois anos foi tempo suficiente para John e Dave apresentarem essa história não como uma atualização aleatória da linha de produto, mas como um ponto de inflexão definidor da tecnologia da computação na última década. Aquela quarta edição teve ênfase reduzida no paralelismo em nível de instrução (Instruction-Level Parallelism – ILP) em favor de um material adicional sobre paralelismo, algo em que a edição atual vai além, dedicando dois capítulos ao paralelismo em nível de thread e dados, enquanto limita a discussão sobre ILP a um único capítulo. Os leitores que estão sendo apresentados aos novos engines de processamento gráfico vão se beneficiar especialmente do novo Capítulo 4, que se concentra no paralelismo de dados, explicando as soluções diferentes mas lentamente convergentes oferecidas pelas extensões multimídia em processadores de uso geral e unidades de processamento gráfico cada vez mais programáveis. De notável relevância prática: se você já lutou com a terminologia CUDA, veja a Figura 4.24 (teaser: a memória compartilhada, na verdade, é local, e a memória global se parece mais com o que você consideraria memória compartilhada).

Embora ainda estejamos no meio dessa mudança para a tecnologia multicore, esta edição abarca o que parece ser a próxima grande mudança: computação em nuvem. Nesse caso, a ubiquidade da conectividade à Internet e a evolução de serviços Web atraentes estão trazendo para o centro do palco dispositivos muito pequenos (smartphones, tablets) e muito grandes (sistemas de computação em escala de depósito). O ARM Cortex A8, uma CPU popular para smartphones, aparece na seção "Juntando tudo" do Capítulo 3, e um Capítulo 6 totalmente novo é dedicado ao paralelismo em nível de requisição e dados no contexto dos sistemas de computação em escala de depósito. Neste novo capítulo, John e Dave apresentam esses novos grandes clusters como uma nova classe distinta de computadores – um convite aberto para os arquitetos de computadores ajudarem a moldar

esse campo emergente. Os leitores vão apreciar o modo como essa área evoluiu na última década, comparando a arquitetura do cluster Google descrita na terceira edição com a encanação mais moderna apresentada no Capítulo 6 desta versão.

Aqueles que estão retomando este livro vão poder apreciar novamente o trabalho de dois destacados cientistas da computação que, ao longo de suas carreiras, aperfeiçoaram a arte de combinar o tratamento das ideias com princípios acadêmicos com uma profunda compreensão dos produtos e tecnologias de ponta dessa indústria. O sucesso dos autores nas interações com a indústria não será uma surpresa para aqueles que testemunharam como Dave conduz seus retiros bianuais de projeto, foruns meticulosamente elaborados para extrair o máximo das colaborações acadêmico-industriais. Aqueles que se lembram do sucesso do empreendimento de John com o MIPS ou esbarraram com ele em um corredor no Google (o que às vezes acontece comigo) também não vão se surpreender.

E talvez o mais importante: leitores novos e antigos vão obter aquilo por que pagaram. O que fez deste livro um clássico duradouro foi o fato de que cada edição não é uma atualização, mas uma extensa revisão que apresenta as informações mais atuais e insights incomparáveis sobre esse campo fascinante e rapidamente mutável. Para mim, depois de vinte anos nessa profissão, ele é também outra oportunidade de experimentar aquela admiração de estudante por dois professores notáveis.

Prefácio

Por que escrevemos este livro

Ao longo das cinco edições deste livro, nosso objetivo tem sido descrever os princípios básicos por detrás dos desenvolvimentos tecnológicos futuros. Nosso entusiasmo com relação às oportunidades em arquitetura de computadores não diminuiu, e repetimos o que dissemos sobre essa área na primeira edição: "Essa não é uma ciência melancólica de máquinas de papel que nunca funcionarão. Não! É uma disciplina de interesse intelectual incisivo, que exige o equilíbrio entre as forças do mercado e o custo-desempenho-potência, levando a gloriosos fracassos e a alguns notáveis sucessos".

O principal objetivo da escrita de nosso primeiro livro era mudar o modo como as pessoas aprendiam e pensavam a respeito da arquitetura de computadores. Acreditamos que esse objetivo ainda é válido e importante. Esse campo está mudando diariamente e precisa ser estudado com exemplos e medidas reais sobre computadores reais, e não simplesmente como uma coleção de definições e projetos que nunca precisarão ser compreendidos. Damos boas-vindas entusiasmadas a todos os que nos acompanharam no passado e também àqueles que estão se juntando a nós agora. De qualquer forma, prometemos o mesmo enfoque quantitativo e a mesma análise de sistemas reais.

Assim como nas versões anteriores, nos esforçamos para elaborar uma nova edição que continuasse a ser relevante tanto para os engenheiros e arquitetos profissionais quanto para aqueles envolvidos em cursos avançados de arquitetura e projetos de computador. Assim como os livros anteriores, esta edição visa desmistificar a arquitetura de computadores com ênfase nas escolhas de custo-benefício-potência e bom projeto de engenharia. Acreditamos que o campo tenha continuado a amadurecer, seguindo para o alicerce quantitativo rigoroso das disciplinas científicas e de engenharia bem estabelecidas.

Esta edição

Declaramos que a quarta edição de *Arquitetura de Computadores: Uma Abordagem Quantitativa* podia ser a mais significativa desde a primeira edição, devido à mudança para chips multicore. O feedback que recebemos dessa vez foi de que o livro havia perdido o foco agudo da primeira edição, cobrindo tudo igualmente, mas sem ênfase nem contexto. Estamos bastante certos de que não se dirá isso da quinta edição.

Nós acreditamos que a maior parte da agitação está nos extremos do tamanho da computação, com os dispositivos pessoais móveis (Personal Mobile Devices – PMDs), como telefones celulares e tablets, como clientes e computadores em escala de depósito oferecendo computação na nuvem como servidores. (Bons observadores devem ter notado a dica sobre computação em nuvem na capa do livro.) Estamos impressionados com o tema comum desses dois extremos em custo, desempenho e eficiência energética, apesar de sua diferença em tamanho. Como resultado, o contexto contínuo em cada capítulo é

a computação para PMDs e para computadores em escala de depósito, e o Capítulo 6 é totalmente novo com relação a esse tópico.

O outro tema é o paralelismo em todas as suas formas. Primeiro identificamos os dois tipos de paralelismo em nível de aplicação no Capítulo 1, o *paralelismo em nível de dados* (Data-Level Parallelism – DLP), que surge por existirem muitos itens de dados que podem ser operados ao mesmo tempo, e o *paralelismo em nível de tarefa* (Task-Level Parallelism – TLP), que surge porque são criadas tarefas que podem operar independentemente e, em grande parte, em paralelo. Então, explicamos os quatro estilos arquitetônicos que exploram DLP e TLP: *paralelismo em nível de instrução* (Instruction-Level Parallelism – ILP) no Capítulo 3; *arquiteturas de vetor* e *unidades de processamento gráfico (GPUs)* no Capítulo 4, que foi escrito para esta edição; *paralelismo em nível de thread* no Capítulo 5; e *paralelismo em nível de requisição* (Request-Level Parallelism – RLP), através de computadores em escala de depósito no Capítulo 6, que também foi escrito para esta edição. Nós deslocamos a hierarquia de memória mais para o início do livro (Capítulo 2) e realocamos o capítulo sobre sistemas de armazenamento no Apêndice D. Estamos particularmente orgulhosos do Capítulo 4, que contém a mais clara e mais detalhada explicação já dada sobre GPUs, e do Capítulo 6, que é a primeira publicação dos detalhes mais recentes de um computador em escala de depósito do Google.

Como nas edições anteriores, os primeiros três apêndices do livro fornecem o conteúdo básico sobre o conjunto de instruções MIPS, hierarquia de memória e pipelining aos leitores que não leram livros como *Computer Organization and Design*. Para manter os custos baixos e ainda assim fornecer material suplementar que seja do interesse de alguns leitores, disponibilizamos mais nove apêndices onlines em inglês na página www.elsevier.com.br/hennessy. Há mais páginas nesses apêndices do que neste livro!

Esta edição dá continuidade à tradição de usar exemplos reais para demonstrar as ideias, e as seções "Juntando tudo" são novas – as desta edição incluem as organizações de pipeline e hierarquia de memória do processador ARM Cortex A8, o processador Intel Core i7, as GPUs NVIDIA GTX-280 e GTX-480, além de um dos computadores em escala de depósito do Google.

Seleção e organização de tópicos

Como nas edições anteriores, usamos uma técnica conservadora para selecionar os tópicos, pois existem muito mais ideias interessantes em campo do que poderia ser abordado de modo razoável em um tratamento de princípios básicos. Nós nos afastamos de um estudo abrangente de cada arquitetura, com que o leitor poderia se deparar por aí. Nossa apresentação enfoca os principais conceitos que podem ser encontrados em qualquer máquina nova. O critério principal continua sendo o da seleção de ideias que foram examinadas e utilizadas com sucesso suficiente para permitir sua discussão em termos quantitativos.

Nossa intenção sempre foi enfocar o material que não estava disponível em formato equivalente em outras fontes, por isso continuamos a enfatizar o conteúdo avançado sempre que possível. Na realidade, neste livro existem vários sistemas cujas descrições não podem ser encontradas na literatura. (Os leitores interessados estritamente em uma introdução mais básica à arquitetura de computadores deverão ler *Organização e projeto de computadores: a interface hardware/software.*)

Visão geral do conteúdo

Nesta edição o Capítulo 1 foi aumentado: ele inclui fórmulas para energia, potência estática, potência dinâmica, custos de circuito integrado, confiabilidade e disponibilidade. Esperamos que esses tópicos possam ser usados ao longo do livro. Além dos princípios

quantitativos clássicos do projeto de computadores e medição de desempenho, a seção PIAT foi atualizada para usar o novo benchmark SPECPower.

Nossa visão é de que hoje a arquitetura do conjunto de instruções está desempenhando um papel inferior ao de 1990, de modo que passamos esse material para o Apêndice A. Ele ainda usa a arquitetura MIPS64 (para uma rápida revisão, um breve resumo do ISA MIPS pode ser encontrado no verso da contracapa). Para os fãs de ISAs, o Apêndice K aborda 10 arquiteturas RISC, o 80x86, o VAX da DEC e o 360/370 da IBM.

Então, prosseguimos com a hierarquia de memória no Capítulo 2, uma vez que é fácil aplicar os princípios de custo-desempenho-energia a esse material e que a memória é um recurso essencial para os demais capítulos. Como na edição anterior, Apêndice B contém uma revisão introdutória dos princípios de cache, que está disponível caso você precise dela. O Capítulo 2 discute 10 otimizações avançadas dos caches. O capítulo inclui máquinas virtuais, que oferecem vantagens em proteção, gerenciamento de software e gerenciamento de hardware, e tem um papel importante na computação na nuvem. Além de abranger as tecnologias SRAM e DRAM, o capítulo inclui material novo sobre a memória Flash. Os exemplos PIAT são o ARM Cortex A8, que é usado em PMDs, e o Intel Core i7, usado em servidores.

O Capítulo 3 aborda a exploração do paralelismo em nível de instrução nos processadores de alto desempenho, incluindo execução superescalar, previsão de desvio, especulação, escalonamento dinâmico e multithreading. Como já mencionamos, o Apêndice C é uma revisão do pipelining, caso você precise dele. O Capítulo 3 também examina os limites do ILP. Assim como no Capítulo 2, os exemplos PIAT são o ARM Cortex A8 e o Intel Core i7. Como a terceira edição continha muito material sobre o Itanium e o VLIW, esse conteúdo foi deslocado para o Apêndice H, indicando nossa opinião de que essa arquitetura não sobreviveu às primeiras pretensões.

A crescente importância das aplicações multimídia, como jogos e processamento de vídeo, também aumentou a relevância das arquiteturas que podem explorar o paralelismo em nível de dados. Há um crescente interesse na computação usando unidades de processamento gráfico (Graphical Processing Units – GPUs). Ainda assim, poucos arquitetos entendem como as GPUs realmente funcionam. Decidimos escrever um novo capítulo em grande parte para desvendar esse novo estilo de arquitetura de computadores. O Capítulo 4 começa com uma introdução às arquiteturas de vetor, que serve de base para a construção de explicações sobre extensões de conjunto de instrução SIMD e GPUS (o Apêndice G traz mais detalhes sobre as arquiteturas de vetor). A seção sobre GPUs foi a mais difícil de escrever – foram feitas muitas tentativas para obter uma descrição precisa que fosse também fácil de entender. Um desafio significativo foi a terminologia. Decidimos usar nossos próprios termos e, ao traduzi-los, estabelecer uma relação entre eles e os termos oficiais da NVIDIA (uma cópia dessa tabela pode ser encontrada no verso das capas). Esse capítulo apresenta o modelo roofline de desempenho, usando-o para comparar o Intel Core i7 e as GPUs NVIDIA GTX 280 e GTX 480. O capítulo também descreve a GPU Tegra 2 para PMDs.

O Capítulo 5 descreve os processadores multicore. Ele explora as arquiteturas de memória simétricas e distribuídas, examinando os princípios organizacionais e o desempenho. Os tópicos de sincronismo e modelos de consistência de memória vêm em seguida. O exemplo é o Intel Core i7.

Como já mencionado, o Capítulo 6 descreve o mais novo tópico em arquitetura de computadores: os computadores em escala de depósito (Warehouse-Scale Computers – WSCs). Com base na ajuda de engenheiros da Amazon Web Services e Google, esse capítulo integra

detalhes sobre projeto, custo e desempenho dos WSCs que poucos arquitetos conhecem. Ele começa com o popular modelo de programação MapReduce antes de descrever a arquitetura e implementação física dos WSCs, incluindo o custo. Os custos nos permitem explicar a emergência da computação em nuvem, porque pode ser mais barato usar WSCs na nuvem do que em seu datacenter local. O exemplo PIAT é uma descrição de um WSC Google que inclui informações publicadas pela primeira vez neste livro.

Isso nos leva aos Apêndices A a L. O Apêndice A aborda os princípios de ISAs, incluindo MIPS64, e o Apêndice K descreve as versões de 64 bits do Alpha, MIPS, PowerPC e SPARC, além de suas extensões de multimídia. Ele inclui também algumas arquiteturas clássicas (80x86, VAX e IBM 360/370) e conjuntos de instruções embutidas populares (ARM, Thumb, SuperH, MIPS16 e Mitsubishi M32R). O Apêndice H está relacionado a esses conteúdos, pois aborda arquiteturas e compiladores para ISAs VLIW.

Como já dissemos, os Apêndices B e C são tutoriais sobre conceitos básicos de pipelining e caching. Os leitores relativamente iniciantes em caching deverão ler o Apêndice B antes do Capítulo 2, e os novos em pipelining deverão ler o Apêndice C antes do Capítulo 3.

O Apêndice D, "Sistemas de Armazenamento", traz uma discussão maior sobre confiabilidade e disponibilidade, um tutorial sobre RAID com uma descrição dos esquemas RAID 6, e estatísticas de falha de sistemas reais raramente encontradas. Ele continua a fornecer uma introdução à teoria das filas e benchmarks de desempenho de E/S. Nós avaliamos o custo, o desempenho e a confiabilidade de um cluster real: o Internet Archive. O exemplo "Juntando tudo" é o arquivador NetApp FAS6000.

O Apêndice E, elaborado por Thomas M. Conte, consolida o material embutido em um só lugar.

O Apêndice F, sobre redes de interconexão, foi revisado por Timothy M. Pinkston e José Duato. O Apêndice G, escrito originalmente por Krste Asanovic, inclui uma descrição dos processadores vetoriais. Esses dois apêndices são parte do melhor material que conhecemos sobre cada tópico.

O Apêndice H descreve VLIW e EPIC, a arquitetura do Itanium.

O Apêndice I descreve as aplicações de processamento paralelo e protocolos de coerência para o multiprocessamento de memória compartilhada em grande escala. O Apêndice J, de David Goldberg, descreve a aritmética de computador.

O Apêndice L agrupa as "Perspectivas históricas e referências" de cada capítulo em um único apêndice. Ele tenta dar o crédito apropriado às ideias presentes em cada capítulo e o contexto histórico de cada invenção. Gostamos de pensar nisso como a apresentação do drama humano do projeto de computador. Ele também dá referências que o aluno de arquitetura pode querer pesquisar. Se você tiver tempo, recomendamos a leitura de alguns dos trabalhos clássicos dessa área, que são mencionados nessas seções. É agradável e educativo ouvir as ideias diretamente de seus criadores. "Perspectivas históricas" foi uma das seções mais populares das edições anteriores.

Navegando pelo texto

Não existe uma ordem melhor para estudar os capítulos e os apêndices, mas todos os leitores deverão começar pelo Capítulo 1. Se você não quiser ler tudo, aqui estão algumas sequências sugeridas:

- *Hierarquia de memória*: Apêndice B, Capítulo 2 e Apêndice D
- *Paralelismo em nível de instrução*: Apêndice C, Capítulo 3, e Apêndice H
- *Paralelismo em nível de dados*: Capítulos 4 e 6, Apêndice G
- *Paralelismo em nível de* thread: Capítulo 5, Apêndices F e I

- *Paralelismo em nível de requisição*: Capítulo 6
- *ISA*: Apêndices A e K

O Apêndice E pode ser lido a qualquer momento, mas pode ser mais bem aproveitado se for lido após as sequências de ISA e cache. O Apêndice J pode ser lido sempre que a aritmética atraí-lo. Você deve ler a parte correspondente ao Apêndice L depois de finalizar cada capítulo.

Estrutura dos capítulos

O material que selecionamos foi organizado em uma estrutura coerente, seguida em todos os capítulos. Começamos explorando as ideias de um capítulo. Essas ideias são seguidas pela seção "Questões cruzadas", que mostra como as ideias abordadas em um capítulo interagem com as dadas em outros capítulos. Isso é seguido pela "Juntando tudo", que une essas ideias, mostrando como elas são usadas em uma máquina real.

Na sequência vem a seção "Falácias e armadilhas", que permite aos leitores aprender com os erros de outros. Mostramos exemplos de enganos comuns e armadilhas arquitetônicas que são difíceis de evitar, mesmo quando você sabe que estão à sua espera. "Falácias e armadilhas" é uma das seções mais populares do livro. Cada capítulo termina com uma seção de "Comentários finais".

Estudos de caso com exercícios

Cada capítulo termina com estudos de caso e exercícios que os acompanham. Criados por especialistas do setor e acadêmicos, os estudos de caso exploram os principais conceitos do capítulo e verificam o conhecimento dos leitores por meio de exercícios cada vez mais desafiadores. Provavelmente, os instrutores vão achar os estudos de caso detalhados e robustos o bastante para permitir que os leitores criem seus próprios exercícios adicionais.

A numeração de cada exercício (<capítulo.seção >) indica a seção de maior relevância para completá-lo. Esperamos que isso ajude os leitores a evitarem exercícios relacionados a alguma seção que ainda não tenham lido, além de fornecer a eles um trecho para revisão. Os exercícios possuem uma classificação para dar aos leitores uma ideia do tempo necessário para concluí-los:

[10] Menos de 5 minutos (para ler e entender)
[15] 5-15 minutos para dar uma resposta completa
[20] 15-20 minutos para dar uma resposta completa
[25] 1 hora para dar uma resposta completa por escrito
[30] Pequeno projeto de programação: menos de 1 dia inteiro de programação
[40] Projeto de programação significativo: 2 semanas
[Discussão] Tópico para discussão com outros

As soluções para estudos de caso e exercícios estarão disponíveis em inglês para os instrutores que se registrarem na página do livro (www.elsevier.com.br/hennessy)

Material complementar

Uma variedade de recursos está disponível online em www.elsevier.com.br/hennessy, incluindo:

- apêndices de referência – alguns com autoria de especialistas sobre o assunto, convidados – abordando diversos tópicos avançados;
- material de perspectivas históricas que explora o desenvolvimento das principais ideias apresentadas em cada um dos capítulos do texto;

- slides para o instrutor em PowerPoint;
- figuras do livro nos formatos PDF, EPS e PPT;
- links para material relacionado na Web;
- lista de erratas.

Novos materiais e links para outros recursos disponíveis na Web serão adicionados regularmente.

Ajudando a melhorar este livro

Finalmente, é possível ganhar dinheiro lendo este livro (Isso é que é custo-desempenho!). Se você ler os "Agradecimentos", a seguir, verá que nos esforçamos muito para corrigir os erros. Como um livro passa por muitas reimpressões, temos a oportunidade de fazer várias correções. Por isso, se você descobrir qualquer bug extra, entre em contato com a editora norte-americana pelo e-mail <ca5comments@mkp.com>.

Comentários finais

Mais uma vez, este livro é resultado de uma verdadeira coautoria: cada um de nós escreveu metade dos capítulos e uma parte igual dos apêndices. Não podemos imaginar quanto tempo teria sido gasto sem alguém fazendo metade do trabalho, servindo de inspiração quando a tarefa parecia sem solução, proporcionando um insight-chave para explicar um conceito difícil, fazendo críticas aos capítulos nos fins de semana e se compadecendo quando o peso de nossas outras obrigações tornava difícil continuar escrevendo (essas obrigações aumentaram exponencialmente com o número de edições, como mostra o minicurriculum de cada um). Assim, mais uma vez, compartilhamos igualmente a responsabilidade pelo que você está para ler.

John Hennessy & David Patterson

CAPÍTULO 1

Fundamentos do projeto e análise quantitativos

"Eu acho justo dizer que os computadores pessoais se tornaram a ferramenta mais poderosa que já criamos. Eles são ferramentas de comunicação, são ferramentas de criatividade e podem ser moldados por seu usuário."

Bill Gates, 24 de fevereiro de 2004

1.1 INTRODUÇÃO

A tecnologia de computação fez um progresso incrível no decorrer dos últimos 65 anos, desde que foi criado o primeiro computador eletrônico de uso geral. Hoje, por menos de US$ 500 se compra um computador pessoal com mais desempenho, mais memória principal e mais armazenamento em disco do que um computador comprado em 1985 por US$ 1 milhão. Essa melhoria rápida vem tanto dos avanços na tecnologia usada para montar computadores quanto da inovação no projeto de computadores.

Embora as melhorias tecnológicas tenham sido bastante estáveis, o progresso advindo de arquiteturas de computador aperfeiçoadas tem sido muito menos consistente. Durante os primeiros 25 anos de existência dos computadores eletrônicos, ambas as forças fizeram uma importante contribuição, promovendo a melhoria de desempenho de cerca de 25% por ano. O final da década de 1970 viu o surgimento do microprocessador. A capacidade do microprocessador de acompanhar as melhorias na tecnologia de circuito integrado

levou a uma taxa de melhoria mais alta — aproximadamente 35% de crescimento por ano, em desempenho.

Essa taxa de crescimento, combinada com as vantagens do custo de um microprocessador produzido em massa, fez com que uma fração cada vez maior do setor de computação fosse baseada nos microprocessadores. Além disso, duas mudanças significativas no mercado de computadores facilitaram, mais do que em qualquer outra época, o sucesso comercial com uma nova arquitetura: 1) a eliminação virtual da programação em linguagem Assembly reduziu a necessidade de compatibilidade de código-objeto; 2) a criação de sistemas operacionais padronizados, independentes do fornecedor, como UNIX e seu clone, o Linux, reduziu o custo e o risco de surgimento de uma nova arquitetura.

Essas mudanças tornaram possível o desenvolvimento bem-sucedido de um novo conjunto de arquiteturas com instruções mais simples, chamadas arquiteturas RISC (Reduced Instruction Set Computer — computador de conjunto de instruções reduzido), no início da década de 1980. As máquinas baseadas em RISC chamaram a atenção dos projetistas para duas técnicas críticas para o desempenho: a exploração do *paralelismo em nível de instrução* (inicialmente por meio do pipelining e depois pela emissão de múltiplas instruções) e o uso de caches (inicialmente em formas simples e depois usando organizações e otimizações mais sofisticadas).

Os computadores baseados em RISC maximizaram o padrão de desempenho, forçando as arquiteturas anteriores a acompanhar esse padrão ou a desaparecer. O Vax da Digital Equipment não fez isso e, por essa razão, foi substituído por uma arquitetura RISC. A Intel acompanhou o desafio, principalmente traduzindo instruções 80x86 (ou IA-32) para instruções tipo RISC, internamente, permitindo a adoção de muitas das inovações pioneiras nos projetos RISC. À medida que a quantidade de transistores aumentava no final dos anos 1990, o overhead do hardware para traduzir a arquitetura x86 mais complexa tornava-se insignificante. Em aplicações específicas, como telefones celulares, o custo com relação à potência e à área de silício relativo ao overhead da tradução do x86 ajudou uma arquitetura RISC, a ARM, a se tornar dominante.

A Figura 1.1 mostra que a combinação de melhorias na organização e na arquitetura dos computadores fez com que o crescimento do desempenho fosse constante durante 17 anos, a uma taxa anual de mais de 50% — ritmo sem precedentes no setor de computação.

Quatro foram os impactos dessa notável taxa de crescimento no século XX. Primeiro, ela melhorou consideravelmente a capacidade disponível aos usuários de computador. Para muitas aplicações, os microprocessadores de desempenho mais alto de hoje ultrapassam o supercomputador de menos de 10 anos atrás.

Em segundo lugar, essa melhoria drástica em custo/desempenho levou a novas classes de computadores. Os computadores pessoais e workstations emergiram nos anos 1980 com a disponibilidade do microprocessador. A última década viu o surgimento dos smartphones e tablets, que muitas pessoas estão usando como plataformas primárias de computação no lugar dos PCs. Esses dispositivos clientes móveis estão usando a internet cada vez mais para acessar depósitos contendo dezenas de milhares de servidores, que estão sendo projetados como se fossem um único gigantesco computador.

Em terceiro lugar, a melhoria contínua da fabricação de semicondutores, como previsto pela lei de Moore, levou à dominância de computadores baseados em microprocessadores por toda a gama de projetos de computador. Os minicomputadores, que tradicionalmente eram feitos a partir de lógica pronta ou de gate arrays, foram substituídos por servidores montados com microprocessadores. Os mainframes foram praticamente substituídos por

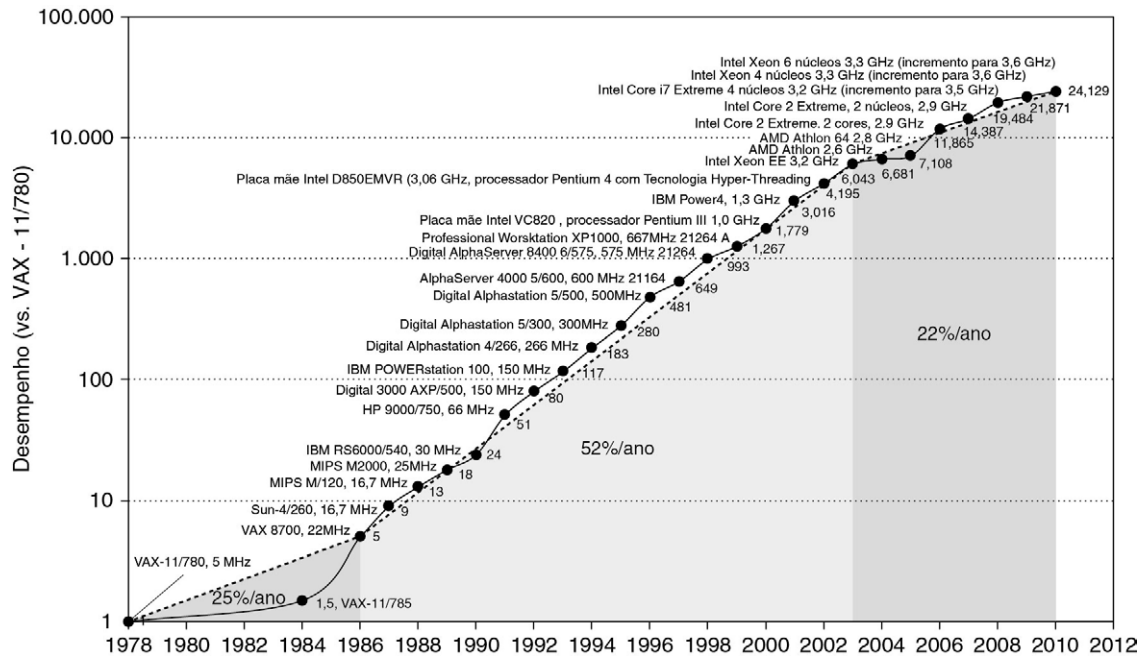



FIGURA 1.1 **Crescimento no desempenho do processador desde o fim da década de 1970.**
Este gráfico mostra o desempenho relativo ao VAX 11/780, medido pelos benchmarks SPECint (Seção 1.8). Antes de meados da década de 1980, o crescimento no desempenho do processador era, em grande parte, controlado pela tecnologia e, em média, era de 25% por ano. O aumento no crescimento, para cerca de 52% desde então, é atribuído a ideias arquitetônicas e organizacionais mais avançadas. Em 2003, esse crescimento levou a uma diferença no desempenho de cerca de um fator de 25 *versus* se tivéssemos continuado com a taxa de 25%. O desempenho para cálculos orientados a ponto flutuante aumentou ainda mais rapidamente. Desde 2003, os limites de potência, paralelismo disponível em nível de instrução e latência longa da memória reduziram o desempenho do uniprocessador para não mais de 22% por ano ou cerca de cinco vezes mais lento do que se tivéssemos continuado com 52% ao ano. (O desempenho SPEC mais rápido desde 2007 teve a paralelização automática ativada, com um número cada vez maior de núcleos por chip a cada ano, então a velocidade do uniprocessador é difícil de medir. Esses resultados se limitam a sistemas de soquete único para reduzir o impacto da paralelização automática.) A Figura 1.11, na página 22, mostra a melhoria nas taxas de clock para essas mesmas três eras. Como o SPEC foi alterado no decorrer dos anos, o desempenho das máquinas mais novas é estimado por um fator de escala que relaciona o desempenho para duas versões diferentes do SPEC (por exemplo, SPEC89, SPEC92, SPEC95, SPEC2000 e SPEC2006).

um pequeno número de microprocessadores encapsulados. Até mesmo os supercomputadores de ponta estão sendo montados com grupos de microprocessadores.

Essas inovações de hardware levaram ao renascimento do projeto de computadores, que enfatizou tanto a inovação arquitetônica quanto o uso eficiente das melhorias da tecnologia. Essa taxa de crescimento foi aumentada de modo que, em 2003, os microprocessadores de alto desempenho eram cerca de 7,5 vezes mais rápidos do que teriam alcançado contando-se apenas com a tecnologia, incluindo a melhoria do projeto do circuito. Ou seja, 52% ao ano *versus* 35% ao ano.

O renascimento do hardware levou ao quarto impacto sobre o desenvolvimento de software. Essa melhoria de 25.000 vezes no desempenho desde 1978 (Fig. 1.1) permitiu aos programadores da atualidade trocar o desempenho pela produtividade. Em vez de utilizar linguagens orientadas ao desempenho, como C e C++, hoje as programações utilizam mais as linguagens, como Java e C#, chamadas de *managed programming languages*. Além do mais, linguagens script, como Python e Ruby, que são ainda mais produtivas, estão ganhando popularidade juntamente com frameworks de programação, como Ruby on Rails. Para manter a produtividade e tentar eliminar o problema do desempenho, os interpretadores com compiladores just-in-time e compilação trace-based estão substituindo os

compiladores e o linkers tradicionais do passado. A implementação de software também está mudando, com o *software como serviço* (Software as a Service — SaaS) usado na internet, substituindo os softwares comprados em uma mídia (shirink-wrapped software), que devem ser instalados e executados em um computador local.

A natureza das aplicações também muda. Fala, som, imagens e vídeo estão tornando-se cada vez mais importantes, juntamente com o tempo de resposta previsível, tão crítico para o usuário. Um exemplo inspirador é o Google Goggles. Esse aplicativo permite apontar a câmera do telefone celular para um objeto e enviar a imagem pela internet sem fio para um computador em escala wharehouse, que reconhece o objeto e dá informações interessantes sobre ele. O aplicativo pode traduzir textos do objeto para outro idioma, ler o código de barras da capa de um livro e dizer se ele está disponível on-line e qual é o seu preço ou, se fizer uma panorâmica com a câmera do celular, dizer quais empresas estão próximas a você, quais são seus sites, números telefônicos e endereços.

Porém, a Figura 1.1 também mostra que esse renascimento de 17 anos acabou. Desde 2003, a melhoria de desempenho dos uniprocessadores únicos caiu para cerca de 22% por ano, devido tanto à dissipação máxima de potência dos chips resfriados a ar como à falta de maior paralelismo no nível de instrução que resta para ser explorado com eficiência. Na realidade, em 2004, a Intel cancelou seus projetos de uniprocessadores de alto desempenho e juntou-se a outras empresas ao mostrar que o caminho para um desempenho mais alto seria através de vários processadores por chip, e não de uniprocessadores mais rápidos.

Isso sinaliza uma passagem histórica, de contar unicamente com o *paralelismo em nível de instrução* (Instruction-Level Parallelism — ILP), foco principal das três primeiras edições deste livro, para contar com o *paralelismo em nível de thread* (Thread-Level Parallelism — TLP) e o *paralelismo em nível de dados* (Data-Level Parallelism — DLP), que são abordados na quarta edição e expandidos nesta. Esta edição também inclui computadores em escala wharehouse. Embora o compilador e o hardware conspirem para explorar o ILP implicitamente sem a atenção do programador, DLP, TLP e RLP são explicitamente paralelos, exigindo a reestruturação do aplicativo para que ele possa explorar o paralelismo explícito. Em alguns casos, isso é fácil. Em muitos, é uma nova grande carga para os programadores.

Este capítulo focaliza as ideias arquitetônicas e as melhorias no compilador que as acompanham e que possibilitaram a incrível taxa de crescimento no século passado, além dos motivos para a surpreendente mudança e os desafios e enfoques promissores iniciais para as ideias arquitetônicas e compiladores para o século XXI. No centro está o enfoque quantitativo para o projeto e a análise de compilador, que usa observações empíricas dos programas, experimentação e simulação como ferramentas. Esse estilo e esse enfoque do projeto de computador são refletidos neste livro. O objetivo, aqui, é estabelecer a base quantitativa na qual os capítulos e apêndices a seguir se baseiam.

Este livro foi escrito não apenas para explorar esse estilo de projeto, mas também para estimulá-lo a contribuir para esse progresso. Acreditamos que essa técnica funcionará para computadores explicitamente paralelos do futuro, assim como funcionou para os computadores implicitamente paralelos do passado.

1.2 CLASSES DE COMPUTADORES

Essas alterações prepararam o palco para uma mudança surpreendente no modo como vemos a computação, nas aplicações computacionais e nos mercados de computadores, neste novo século. Nunca, desde a criação do computador pessoal, vimos mudanças tão notáveis em como os computadores se parecem e como são usados. Essas mudanças no uso

ELSEVIER

Recurso	Dispositivo pessoal móvel (PMD)	Desktop	Servidor	Clusters/computador de armazenamento em escala	Embarcados
Preço do sistema	\$100-\$1.000	\$300-\$2.500	\$5.000-\$10.000.000	\$100.000-\$200.000.000	\$10-\$100.000
Preço do microprocessador	\$10-\$100	\$50-\$500	\$200-\$2.000	\$50-\$250	\$0,01-\$100
Questões críticas de projeto do sistema	Custo, energia, desempenho de mídia, capacidade de resposta	Preço-desempenho, consumo de energia, desempenho de gráficos	Throughput, disponibilidade, escalabilidade, energia	Preço-desempenho, throughput, proporcionalidade de energia	Preço, consumo de energia, desempenho específico da aplicação

FIGURA 1.2 Um resumo das cinco classes de computação principais e suas características de sistema.
As vendas em 2010 incluíram cerca de 1,8 bilhão de PMDs (90% deles em telefones celulares), 350 milhões de PCs desktop e 20 milhões de servidores. O número total de processadores embarcados vendidos foi de quase 19 bilhões. No total, 6,1 bilhões de chips baseados em tecnologia ARM foram vendidos em 2010. Observe a ampla faixa de preços de servidores e sistemas embarcados, que vão de pendrives USB a roteadores de rede. Para servidores, essa faixa varia da necessidade de sistemas multiprocessadores com escala muito ampla ao processamento de transações de alto nível.

do computador geraram três mercados de computador diferentes, cada qual caracterizado por diferentes aplicações, requisitos e tecnologias de computação. A Figura 1.2 resume essas classes principais de ambientes de computador e suas características importantes.

Dispositivo pessoal móvel (PMD)

Dispositivo pessoal móvel (Personal Mobile Device — PMD) é o nome que aplicamos a uma coleção de dispositivos sem fio com interfaces de usuário multimídia, como telefones celulares, tablets, e assim por diante. O custo é a principal preocupação, dado que o preço para o consumidor de todo o produto é de algumas centenas de dólares. Embora a ênfase na eficiência energética seja frequentemente orientada pelo uso de baterias, a necessidade de usar materiais menos caros — plástico em vez de cerâmica — e a ausência de uma ventoinha para resfriamento também limitam o consumo total de energia. Examinamos a questão da energia e da potência em detalhes na Seção 1.5. Aplicativos para PMDs muitas vezes são baseados na web e orientados para a mídia, como no exemplo acima (Google Goggles). Os requisitos de energia e tamanho levam ao uso de memória Flash para armazenamento (Cap. 2) no lugar de discos magnéticos.

A capacidade de resposta e previsibilidade são características-chave para aplicações de mídia. Um requisito de *desempenho em tempo real* significa que um segmento da aplicação tem um tempo absoluto máximo de execução. Por exemplo, ao se reproduzir vídeo em um PMD, o tempo para processar cada quadro de vídeo é limitado, pois o processador precisa aceitar e processar o próximo quadro rapidamente. Em algumas aplicações, existe um requisito mais sutil: o tempo médio para determinada tarefa é restrito, tanto quanto o número de ocorrências quando um tempo máximo é ultrapassado. Essas técnicas, também chamadas *tempo real flexível*, são necessárias quando é possível perder, ocasionalmente, a restrição de tempo em um evento, desde que não haja muita perda. O desempenho em tempo real costuma ser altamente dependente da aplicação.

Outras características-chave em muitas aplicações PMD são a necessidade de minimizar a memória e a necessidade de minimizar o consumo de potência. A eficiência energética é orientada tanto pela potência da bateria quanto pela dissipação de calor. A memória pode ser uma parte substancial do custo do sistema, e é importante otimizar o tamanho dessa memória nesses casos. A importância do tamanho da memória é traduzida com ênfase no tamanho do código, pois o tamanho dos dados é ditado pela aplicação.

Computação de desktop

O primeiro e maior mercado em termos financeiros ainda é o de computadores desktop. A computação desktop varia desde sistemas inferiores, vendidos por menos de US$ 300, até estações de trabalho de ponta altamente configuradas, que podem custar US$ 2.500. Desde 2008, mais da metade dos computadores desktops fabricados, por ano, corresponde a computadores laptop alimentados por bateria.

Por todo esse intervalo de preço e capacidade, o mercado de desktop costuma ser orientado a otimizar a relação *preço-desempenho*. Essa combinação de desempenho (medido principalmente em termos de desempenho de cálculo e desempenho de gráficos) e preço de um sistema é o que mais importa para os clientes nesse mercado e, portanto, para os projetistas de computadores. Como resultado, os microprocessadores mais novos, de desempenho mais alto, e os microprocessadores de custo reduzido normalmente aparecem primeiro nos sistemas de desktop (ver, na Seção 1.6, uma análise das questões que afetam o custo dos computadores).

A computação de desktop também costuma ser razoavelmente bem caracterizada em termos de aplicações e benchmarking, embora o uso crescente de aplicações centradas na web, interativas, imponha novos desafios na avaliação do desempenho.

Servidores

Com a passagem para a computação desktop nos anos 1980, o papel dos servidores cresceu para oferecer serviços de arquivo e computação em maior escala e mais seguros. Tais servidores se tornaram a espinha dorsal da computação empresarial de alta escala, substituindo o mainframe tradicional.

Para os servidores, diferentes características são importantes. Primeiro, a disponibilidade é crítica (discutimos a dependência na Seção 1.7). Considere os servidores que suportam as máquinas de caixa eletrônico para bancos ou os sistemas de reserva de linhas aéreas. As falhas desses sistemas de servidor são muito mais catastróficas do que as falhas de um único desktop, pois esses servidores precisam operar sete dias por semana, 24 horas por dia. A Figura 1.3 estima as perdas de receita em função do tempo de paralisação para aplicações de servidor.

Aplicação	Custo de tempo de paralisação por hora (milhares de $)	Perdas anuais (milhões de $) com tempo de paralisação de		
		1% (87,6 h/ano)	0,5% (43,8 h/ano)	0,1% (8,8 h/ano)
Operações de corretagem	$ 6.450	$ 565	$ 283	$ 56,5
Autorização de cartão de crédito	$ 2.600	$ 228	$ 114	$ 22,8
Serviços de remessa de pacotes	$ 150	$ 13	$ 6,6	$ 1,3
Canal de compras domésticas	$ 113	$ 9,9	$ 4,9	$ 1,0
Centro de vendas por catálogo	$ 90	$ 7,9	$ 3,9	$ 0,8
Central de reserva aérea	$ 89	$ 7,9	$ 3,9	$ 0,8
Ativação de serviço de celular	$ 41	$ 3,6	$ 1,8	$ 0,4
Taxas de rede on-line	$ 25	$ 2,2	$ 1,1	$ 0,2
Taxas de serviço de caixa eletrônico	$ 14	$ 1,2	$ 0,6	$ 0,1

FIGURA 1.3 Os custos arredondados para o milhar mais próximo de um sistema não disponível são mostrados com uma análise do custo do tempo de paralisação (em termos de receita perdida imediatamente), considerando três níveis de disponibilidade diferentes e que o tempo de paralisação é distribuído uniformemente.

Esses dados são de Kembel (2000) e foram coletados e analisados pela Contingency Planning Research.

Por fim, os servidores são projetados para um throughput eficiente. Ou seja, o desempenho geral do servidor — em termos de transações por minuto ou páginas web atendidas por segundo — é o fator crucial. A capacidade de resposta a uma solicitação individual continua sendo importante, mas a eficiência geral e a eficiência de custo, determinadas por quantas solicitações podem ser tratadas em uma unidade de tempo, são as principais métricas para a maioria dos servidores. Retornamos à questão de avaliar o desempenho para diferentes tipos de ambientes de computação na Seção 1.8.

Computadores clusters/escala wharehouse

O crescimento do software como serviço (Software as a Service — SaaS) para aplicações como busca, redes sociais, compartilhamento de vídeo, games multiplayer, compras on-line, e assim por diante, levou ao crescimento de uma classe de computadores chamados *clusters*. Clusters são coleções de computadores desktop ou servidores conectados por redes locais para funcionar como um único grande computador. Cada nó executa seu próximo sistema operacional, e os nós se comunicam usando um protocolo de rede. Os maiores clusters são chamados *computadores de armazenamento em escala* (Warehouse-Scale Computers — WSCs), uma vez que eles são projetados para que dezenas de milhares de servidores possam funcionar como um só. O Capítulo 6 descreve essa classe de computadores extremamente grandes.

A relação preço-desempenho e o consumo de potência são críticos para os WSCs, já que eles são tão grandes. Como o Capítulo 6 explica, 80% do custo de US$ 90 milhões de um WSC é associado à potência e ao resfriamento interior dos computadores. Os próprios computadores e o equipamento de rede custam outros US$ 70 milhões e devem ser substituídos após alguns anos de uso. Ao comprar tanta computação, você precisa fazer isso com sabedoria, já que uma melhoria de 10% no desempenho de preço significa uma economia de US$ 7 milhões (10% de 70 milhões).

Os WSCs estão relacionados com os servidores no sentido de que a disponibilidade é crítica. Por exemplo, a Amazon.com teve US$ 13 bilhões de vendas no quarto trimestre de 2010. Como em um trimestre há cerca de 2.200 horas, a receita média por hora foi de quase US$ 6 milhões. Durante uma hora de pico de compras no Natal, a perda potencial seria muitas vezes maior. Como explicado no Capítulo 6, a diferença em relação aos servidores é que os WSCs usam componentes redundantes baratos, como building blocks, confiando em uma camada de software para capturar e isolar as muitas falhas que vão ocorrer com a computação nessa escala. Note que a escalabilidade para um WSC é tratada pela rede LAN que conecta os computadores, e não por um hardware integrado de computador, como no caso dos servidores.

Uma categoria relacionada comos WSCs é a dos *supercomputadores*, que custam dezenas de milhões de dólares, mas os supercomputadores são diferentes, pois enfatizam o desempenho em ponto flutuante e, a cada vez, executam programas em lotes grandes, com comunicação pesada, por semanas. Esse acoplamento rígido leva ao uso de redes internas muito mais rápidas. Em contraste, os WSCs enfatizam aplicações interativas, armazenamento em grande escala, dependência e grande largura de banda de internet.

Computadores embarcados

Os computadores embarcados são encontrados em máquinas do dia a dia: fornos de micro-ondas, máquinas de lavar, a maioria das impressoras, switches de rede e todos os carros contêm microprocessadores embarcados simples.

Muitas vezes, os processadores em um PMD são considerados computadores embarcados, mas os estamos colocando em uma categoria separada, porque os PMDs são plataformas que podem executar softwares desenvolvidos externamente e compartilham muitas das

características dos computadores desktop. Outros dispositivos embarcados são mais limitados em sofisticação de hardware e software. Nós usamos a capacidade de executar software de terceiros como a linha divisória entre computadores embarcados e não embarcados.

Os computadores embarcados possuem a mais extensa gama de poder de processamento e custo. Eles incluem processadores de 8 e 16 bits, que podem custar menos de 10 centavos de dólar, microprocessadores de 32 bits, que executam 100 milhões de instruções por segundo e custam menos de US$ 5, e processadores de ponta para switches de rede mais recentes, que custam US$ 100 e podem executar bilhões de instruções por segundo. Embora a gama da capacidade de computação no mercado de computação embarcada seja muito extensa, o preço é um fator importante no projeto de computadores para esse espaço. Existem requisitos de desempenho, é claro, mas o objetivo principal normalmente é atender a necessidade de desempenho a um preço mínimo, em vez de conseguir desempenho mais alto a um preço mais alto.

A maior parte deste livro se aplica ao projeto, uso e desempenho de processadores embarcados, sejam eles microprocessadores encapsulados, sejam núcleos de microprocessadores que serão montados com outro hardware de uso específico. Na realidade, a terceira edição deste livro incluiu exemplos de computação embarcada para ilustrar as ideias em cada capítulo.

Infelizmente, a maioria dos leitores considerou esses exemplos insatisfatórios, pois os dados levam ao projeto quantitativo e à avaliação de computadores desktop, e servidores ainda não foram bem estendidos para a computação embarcada (ver os desafios com o EEMBC, por exemplo, na Seção 1.8). Portanto, por enquanto ficamos com as descrições qualitativas, que não se ajustam bem ao restante do livro. Como resultado, nesta edição, consolidamos o material embarcado em um único novo apêndice. Acreditamos que o Apêndice E melhore o fluxo de ideias no texto, permitindo ainda que os leitores percebam como os diferentes requisitos afetam a computação embarcada.

Classes de paralelismo e arquiteturas paralelas

Paralelismo em múltiplos níveis é a força impulsionadora do projeto de computadores pelas quatro classes de computadores, tendo a energia e o custo como as principais restrições. Existem basicamente dois tipos de paralelismo em aplicações:

1. *Paralelismo em nível de dados* (Data-Level Parallelism — DLP): surge porque existem muitos itens de dados que podem ser operados ao mesmo tempo.
2. *Paralelismo em nível de tarefas* (Task-Level Parallelism — TLP): surge porque são criadas tarefas que podem operar de modo independente e principalmente em paralelo.

O hardware do computador pode explorar esses dois tipos de paralelismo de aplicação de quatro modos principais:

1. O *paralelismo em nível de instruções* explora o paralelismo em nível de dados a níveis modestos com auxílio do compilador, usando ideias como pipelining e em níveis médios usando ideias como execução especulativa.
2. As *arquiteturas vetoriais e as unidades de processador gráfico* (Graphic Processor Units — GPUs) exploram o paralelismo em nível de dados aplicando uma única instrução a uma coleção de dados em paralelo.
3. O *paralelismo em nível de thread* explora o paralelismo em nível de dados ou o paralelismo em nível de tarefas em um modelo de hardware fortemente acoplado, que permite a interação entre threads paralelos.
4. O *paralelismo em nível de requisição* explora o paralelismo entre tarefas muito desacopladas especificadas pelo programador ou pelo sistema operacional.

Esses quatro modos de o hardware suportar o paralelismo em nível de dados e o paralelismo em nível de tarefas têm 50 anos. Quando Michael Flynn (1966) estudou os esforços de computação paralela nos anos 1960, encontrou uma classificação simples cujas abreviações ainda usamos hoje. Ele examinou o paralelismo nos fluxos de instrução e dados chamados pelas instruções no componente mais restrito do multiprocessador, colocando todos os computadores em uma de quatro categorias:

1. *Fluxo simples de instrução, fluxo simples de dados* (Single Instruction Stream, Single Data Stream — SISD). Essa categoria é o uniprocessador. O programador pensa nela como o computador sequencial padrão, mas ele pode explorar o paralelismo em nível de instrução. O Capítulo 3 cobre as arquiteturas SISD que usam técnicas ILP, como a execução superescalar e a execução especulativa.
2. *Fluxo simples de instrução, fluxos múltiplos de dados* (Single Instruction Stream, Multiple Data Streams — SIMD). A mesma instrução é executada por múltiplos processadores usando diferentes fluxos de dados. *Computadores* SIMD exploram o *paralelismo em nível de dados* ao aplicar as mesmas operações a múltiplos itens em paralelo. Cada processador tem sua própria memória de dados (daí o MD de SIMD), mas existe uma única memória de instruções e um único processador de controle, que busca e envia instruções. O Capítulo 4 cobre o DLP e três diferentes arquiteturas que o exploram: arquiteturas vetoriais, extensões multimídia a conjuntos de instruções-padrão e GPUs.
3. *Fluxos de múltiplas instruções, fluxo simples de dados* (Multiple Instruction Stream, Single Data Stream — MISD). Nenhum microprocessador comercial desse tipo foi construído até hoje, mas ele completa essa classificação simples.
4. *Fluxos múltiplos de instruções, fluxos múltiplos de dados* (Multiple Instruction Streams, Multiple Data Streams — MIMD). Cada processador busca suas próprias instruções e opera seus próprios dados, buscando o paralelismo em nível de tarefa. Em geral, o MIMD é mais flexível do que o SIMD e, por isso, em geral é mais aplicável, mas é inerentemente mais caro do que o SIMD. Por exemplo, computadores MIMD podem também explorar o paralelismo em nível de dados, embora o overhead provavelmente seja maior do que seria visto em um computador SIMD. Esse overhead significa que o tamanho do grão deve ser suficientemente grande para explorar o paralelismo com eficiência. O Capítulo 5 cobre arquiteturas MIMD fortemente acopladas que exploram o *paralelismo em nível de thread*, uma vez que múltiplos threads em cooperação operam em paralelo. O Capítulo 6 cobre arquiteturas MIMD fracamente acopladas — especificamente, *clusters* e *computadores em escala* — que exploram o *paralelismo em nível de requisição*, em que muitas tarefas independentes podem ocorrer naturalmente em paralelo, com pouca necessidade de comunicação ou sincronização.

Essa taxonomia é um modelo grosseiro, já que muitos processadores paralelos são híbridos das classes SISD, SIMD e MIMD. Mesmo assim, é útil colocar um framework no espaço de projeto para os computadores que veremos neste livro.

1.3 DEFINIÇÃO DA ARQUITETURA DO COMPUTADOR

A tarefa que o projetista de computador desempenha é complexa: determinar quais atributos são importantes para um novo computador, depois projetar um computador para maximizar o desempenho enquanto permanece dentro das restrições de custo, potência e disponibilidade. Essa tarefa possui muitos aspectos, incluindo o projeto do conjunto de instruções, a organização funcional, o projeto lógico e a implementação. A implementação pode abranger o projeto do circuito integrado, o acondicionamento, a

potência e o resfriamento. A otimização do projeto requer familiaridade com uma gama de tecnologias muito extensa, desde compiladores e sistemas operacionais até o projeto lógico e o o acondicionamento.

No passado, o nome *arquitetura de computadores* normalmente se referia apenas ao projeto do conjunto de instruções. Outros aspectos do projeto de computadores eram chamados de *implementação*, normalmente insinuando que a implementação não é interessante ou é menos desafiadora.

Acreditamos que essa visão seja incorreta. A tarefa do arquiteto ou do projetista é muito mais do que projetar o conjunto de instruções, e os obstáculos técnicos nos outros aspectos do projeto provavelmente são mais desafiadores do que aqueles encontrados no projeto do conjunto de instruções. Veremos rapidamente a arquitetura do conjunto de instruções antes de descrever os desafios maiores para o arquiteto de computador.

Arquitetura do conjunto de instruções

Neste livro, usamos o nome *arquitetura do conjunto de instruções* (Instruction Set Architecture — ISA) para nos referir ao conjunto de instruções visíveis pelo programador. A arquitetura do conjunto de instruções serve como interface entre o software e o hardware. Essa revisão rápida da *arquitetura do conjunto de instruções* usará exemplos do 80x86, do ARM e do MIPS para ilustrar as sete dimensões de uma *arquitetura do conjunto de instruções*. Os Apêndices A e K oferecem mais detalhes sobre as três *arquiteturas de conjunto de instruções*.

1. *Classes de ISA*. Hoje, quase todas as arquiteturas de conjunto de instruções são classificadas como arquiteturas de registradores de propósito geral (GPRs), em que os operandos são registradores ou locais de memória. O 80x86 contém 16 registradores de propósito geral e 16 que podem manter dados de ponto flutuante (FPRs), enquanto o MIPS contém 32 registradores de propósito geral e 32 de ponto flutuante (Fig. 1.4). As duas versões populares dessa classe são arquiteturas de conjunto de instruções *registrador-memória*, como o 80x86,

Nome	Número	Uso	Preservado entre uma chamada?
`$zero`	0	Valor constante 0	N.D.
`$at`	1	Temporário do assembler	Não
`$v0-$v1`	2-3	Valores para resultados de função e avaliação de expressão	Não
`$a0-$a3`	4-7	Argumentos	Não
`$t0-$t7`	8-15	Temporários	Não
`$s0-$s7`	16-23	Temporários salvos	Sim
`$t8-$t9`	24-25	Temporários	Não
`$k0-$k1`	26-27	Reservado para kernel do SO	Não
`$gp`	28	Ponteiro global	Sim
`$sp`	29	Ponteiro de pilha	Sim
`$fp`	30	Ponteiro de frame	Sim
`$ra`	31	Endereço de retorno	Sim

FIGURA 1.4 Registradores do MIPS e convenções de uso.
Além dos 32 registradores de propósito geral (R0-R31), o MIPS contém 32 registradores de ponto flutuante (F0-F31) que podem manter um número de precisão simples de 32 bits ou um número de precisão dupla de 64 bits.

que podem acessar a memória como parte de muitas instruções, e arquiteturas de conjunto de instruções *load-store*, como o MIPS, que só podem acessar a memória com instruções load ou store. Todas as arquiteturas de conjunto de instruções recentes são *load-store*.

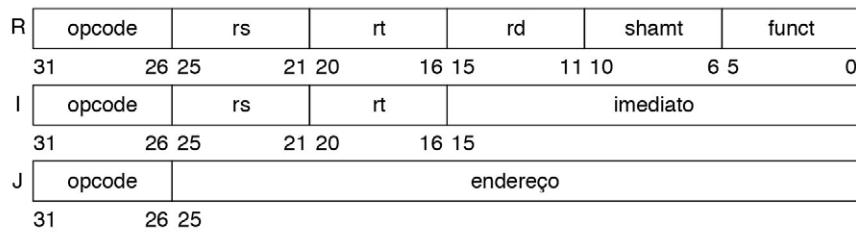
2. *Endereçamento de memória.* Praticamente todos os computadores desktop e servidores, incluindo o 80x86 e o MIPS, utilizam endereçamento de byte para acessar operandos da memória. Algumas arquiteturas, como ARM e MIPS, exigem que os objetos estejam *alinhados*. Um acesso a um objeto com tamanho de *s* bytes no endereço de byte *A* está alinhado se *A* mod *s* = 0 (Fig. A.5 na página A-7.) O 80x86 não exige alinhamento, mas os acessos geralmente são mais rápidos se os operandos estiverem alinhados.
3. *Modos de endereçamento.* Além de especificar registradores e operandos constantes, os modos de endereçamento especificam o endereço de um objeto na memória. Os modos de endereçamento do MIPS são *registrador*, *imediato* (para constantes) e *deslocamento*, em que um deslocamento constante é acrescentado a um registrador para formar o endereço da memória. O 80x86 suporta esses três modos de endereçamento e mais três variações de deslocamento: nenhum registrador (absoluto), dois registradores (indexados pela base com deslocamento) e dois registradores em que um registrador é multiplicado pelo tamanho do operando em bytes (base com índice em escala e deslocamento). Ele contém mais dos três últimos, sem o campo de deslocamento, mais o indireto por registrador, indexado e base com índice em escala. O ARM tem os três modos de endereçamento MIPS mais o endereçamento relativo a PC, a soma de dois registradores e a soma de dois registradores em que um registrador é multiplicado pelo tamanho do operando em bytes. Ele também tem endereçamento por autoincremento e autodecremento, em que o endereço calculado substitui o conteúdo de um dos registradores usados para formar o endereço.
4. *Tipos e tamanhos de operandos.* Assim como a maioria das arquiteturas de conjunto de instruções, o MIPS, o ARM e o 80x86 admitem tamanhos de operando de 8 bits (caractere ASCII), 16 bits (caractere Unicode ou meia palavra), 32 bits (inteiro ou palavra), 64 bits (dupla palavra ou inteiro longo) e ponto flutuante IEEE 754 com 32 bits (precisão simples) e 64 bits (precisão dupla). O 80x86 também admite ponto flutuante de 80 bits (precisão dupla estendida).
5. *Operações.* As categorias gerais de operações são transferência de dados, lógica e aritmética, controle (analisado em seguida) e ponto flutuante. O MIPS é uma arquitetura de conjunto de instruções simples e fáceis de executar em um pipeline, representando as arquiteturas RISC usadas em 2011. A Figura 1.5 resume a arquitetura do conjunto de instruções do MIPS. O 80x86 possui um conjunto de operações maior e muito mais rico (Apêndice K).
6. *Instruções de fluxo de controle.* Praticamente todas as arquiteuras de conjunto de instruções, incluindo essas três, admitem desvios condicionais, saltos incondicionais, chamadas e retornos de procedimento. As três usam endereçamento relativo ao PC, no qual o endereço de desvio é especificado por um campo de endereço que é somado ao PC. Existem algumas pequenas diferenças. Desvios condicionais do MIPS (BE, BNE etc.) testam o conteúdo dos registradores, enquanto os desvios do 80x86 e ARM testam o conjunto de bits de código de condição como efeitos colaterais das operações aritméticas/lógicas. As chamadas de procedimento do ARM e MIPS colocam o endereço de retorno em um registrador, enquanto a chamada do 80x86 (CALLF) coloca o endereço de retorno em uma pilha na memória.
7. *Codificando uma arquitetura de conjunto de instruções.* Existem duas opções básicas na codificação: *tamanho fixo* e *tamanho variável*. Todas as instruções do ARM e MIPS possuem 32 bits de extensão, o que simplifica a decodificação da instrução.

Tipo de instrução/opcode	Significado da instrução
Transferências de dados	*Move dados entre registradores e memória ou entre o inteiro e PF ou registradores especiais; somente o modo de endereço de memória é deslocamento de 16 bits + conteúdo de um GPR*
LB, LBU, SB	Load byte, load byte unsigned, store byte (de/para registradores inteiros)
LH, LHU, SH	Load half word, load half word unsigned, store half word (de/para registradores inteiros)
LW, LWU, SW	Load word, load word unsigned, store word (de/para registradores inteiros)
LD, SD	Load double word, store double word (de/para registradores inteiros)
L.S, L.D, S.S, S.D	Load SP float, load DP float, store SP float, store DP float
MFC0, MTC0	Copia de/para GPR para/de um registrador especial
MOV.S, MOV.D	Copia um registrador de PF (ponto flutuante) SP ou DP para outro registrador de PF
MFC1, MTC1	Copia 32 bits de/para registradores de PFpara/de registradores inteiros
Aritmética/lógica	*Operações sobre dados inteiros ou lógicos em GPRs; aritmética com sinal trap de overflow*
DADD, DADDI, DADDU, DADDIU	Add, add immediate (todos os imediatos são de 16 bits); com e sem sinal
DSUB, DSUBU	Subtração; com e sem sinal
DMUL, DMULU, DDIV, DDIVU, MADD	Multiplicação e divisão, com e sem sinal; multiply-add; todas as operações apanham e geram valores de 64 bits
AND, ANDI	And, and immediate
OR, ORI, XOR, XORI	Or, or immediate, exclusive or, exclusive or immediate
LUI	Load upper immediate; carrega bits 32 a 47 do registrador com imediato, depois estende o sinal
DSLL, DSRL, DSRA, DSLLV, DSRLV, DSRAV	Deslocamentos: tanto imediato (DS__) quanto variável (DS__V); deslocamentos são shift left logical, right logical, right arithmetic
SLT, SLTI, SLTU, SLTIU	Set less than, set less than immediate; com e sem sinal
Controle	*Desvios condicionais e saltos; relativo ao PC ou através do registrador*
BEQZ, BNEZ	Desvia se GPRs forem iguais/não iguais a zero; offset de 16 bits a partir de PC + 4
BEQ, BNE	Desvia se GPR forem iguais/não iguais; offset de 16 bits a partir de PC + 4
BC1T, BC1F	Testa bit de comparação no registrador de *status* de PF e desvia; offset de 16 bits a partir de PC + 4
MOVN, MOVZ	Copia GPR para outro GPR se terceiro GPR for negativo, zero
J, JR	Salto: offset de 26 bits a partir de PC + 4 (J) ou destino no registrador (JR)
JAL, JALR	Salto e link: salva PC + 4 em R31, destino é relativo ao PC (JAL) ou a um registrador (JALR)
TRAP	Transferência para sistema operacional em um endereço de vetor
ERET	Retorna ao código do usuário a partir de uma exceção; restaura modo do usuário
Ponto flutuante	*Operações de PF nos formatos DP e SP*
ADD.D, ADD.S, ADD.PS	Soma DP, números de SP e pares de números de SP
SUB.D, SUB.S, SUB.PS	Subtrai DP, números de SP e pares de números de SP
MUL.D, MUL.S, MUL.PS	Multiplica DP, ponto flutuante SP e pares de números de SP
MADD.D, MADD.S, MADD.PS	Multiplica-soma DP, números de SP e pares de números SP
DIV.D, DIV.S, DIV.PS	Divide DP, ponto flutuante de SP e pares de números de SP
CVT._._	Converte instruções: CVT.x.y converte do tipo x para tipo y, onde x e y são L (inteiro de 64 bits), W (inteiro de 32 bits), D (DP) ou S (SP). Ambos os operandos são FPRs.
C.__.D, C.__.S	Comparações de DP e SP: "__" = LT, GT, LE, GE, EQ, NE; marca bit no registrador de *status* de PF

FIGURA 1.5 Subconjunto das instruções no MIPS64.

SP = single precision (precisão simples), DP = double precision (precisão dupla). O Apêndice A contém detalhes sobre o MIPS64. Para os dados, o número do bit mais significativo é 0; o menos significativo é 63.

Formatos de instrução básicos

R	opcode	rs	rt	rd	shamt	funct
	31 26	25 21	20 16	15 11	10 6	5 0
I	opcode	rs	rt	imediato		
	31 26	25 21	20 16	15		
J	opcode	endereço				
	31 26	25				

Formatos de instrução de ponto flutuante

FR	opcode	fmt	ft	fs	fd	funct
	31 26	25 21	20 16	15 11	10 6	5 0
FI	opcode	fmt	ft	imediato		
	31 26	25 21	20 16	15		

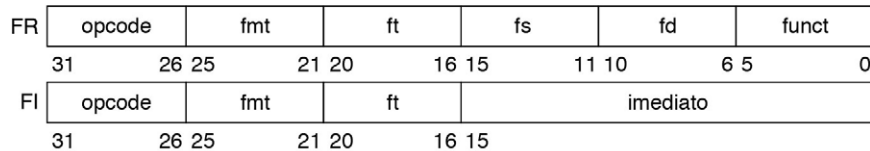



FIGURA 1.6 **Formatos de arquitetura do conjunto de instruções MIPS64.**
Todas as instruções possuem 32 bits de extensão. O formato R é para operações registrador para registrador inteiro, como DADDU, DSUBU, e assim por diante. O formato I é para transferências de dados, desvios e instruções imediatas, como LD, SD, BEQZ e DADDIs. O formato J é para saltos, o formato FR é para operações de ponto flutuante, e o formato FI é para desvios em ponto flutuante.

A Figura 1.6 mostra os formatos de instruções do MIPS. A codificação do 80x86 tem tamanho variável de 1-18 bytes. As instruções de tamanho variável podem ocupar menos espaço que as instruções de tamanho fixo, de modo que um programa compilado para o 80x86 normalmente é menor que o mesmo programa compilado para MIPS. Observe que as opções mencionadas anteriormente afetarão o modo como as instruções são codificadas em uma representação binária. Por exemplo, o número de registradores e o número de modos de endereçamento possuem impacto significativo sobre o tamanho das instruções, pois o campo de registrador e o campo de modo de endereçamento podem aparecer muitas vezes em uma única instrução. (Observe que o ARM e o MIPS, mais tarde, ofereceram extensões para fornecer instruções com 16 bits de extensão para reduzir o tamanho do programa, chamado Thumb ou Thumb-2 e MIPS16, respectivamente.)

No presente, os outros desafios enfrentados pelo arquiteto de computador, além do projeto da arquitetura do conjunto de instruções, são particularmente críticos quando as diferenças entre os conjuntos de instruções são pequenas e existem áreas de aplicação distintas. Portanto, a partir da última edição, o núcleo do material do conjunto de instruções, além dessa revisão rápida, pode ser encontrado nos apêndices (Apêndices A e K).

Neste livro, usamos um subconjunto do MIPS64 como exemplo de arquitetura do conjunto de instruções porque ele é tanto dominante para redes quanto um exemplo elegante das arquiteturas RISC mencionadas, das quais o ARM (Advanced RISC Machine) é o exemplo mais popular. Os processadores ARM estavam em 6,1 bilhões de chips fabricados em 2010, ou aproximadamente 20 vezes o número de chips produzidos de processadores 80x86.

Arquitetura genuína de computador: projetando a organização e o hardware para atender objetivos e requisitos funcionais

A implementação de um computador possui dois componentes: organização e hardware. O termo *organização* inclui os aspectos de alto nível do projeto de um computador, como o sistema de memória, a interconexão de memória e o projeto do processador interno ou CPU (unidade central de processamento, na qual são implementados a aritmética, a

lógica, os desvios e as transferências de dados). O termo *microarquitetura* também é usado no lugar de *organização*. Por exemplo, dois processadores com as mesmas arquiteturas de conjunto de instruções, mas com organizações diferentes, são o AMD Opteron e o Intel Core i7. Ambos implementam o conjunto de instruções x86, mas possuem organizações de pipeline e cache muito diferentes.

A mudança para os processadores múltiplos de microprocessadores levou ao termo *core* usado também para processador. Em vez de se dizer *microprocessador multiprocessador*, o termo *multicore* foi adotado. Dado que quase todos os chips têm múltiplos processadores, o nome *unidade central de processamento*, ou *CPU*, está tornando-se popular.

Hardware refere-se aos detalhes específicos de um computador, incluindo o projeto lógico detalhado e a tecnologia de encapsulamento. Normalmente, uma linha de computadores contém máquinas com arquiteturas de conjunto de instruções idênticas e organizações quase idênticas, diferindo na implementação detalhada do hardware. Por exemplo, o Intel Core i7 (Cap. 3) e o Intel Xeon 7560 (Cap. 5) são praticamente idênticos, mas oferecem taxas de clock e sistemas de memória diferentes, tornando o Xeon 7560 mais eficiente para computadores mais inferiores.

Neste livro, a palavra *arquitetura* abrange os três aspectos do projeto de computadores: arquitetura do conjunto de instruções, organização e hardware.

Os arquitetos dessa área precisam projetar um computador para atender aos requisitos funcionais e também aos objetivos relacionados com preço, potência, desempenho e disponibilidade. A Figura 1.7 resume os requisitos a considerar no projeto de um novo computador. Normalmente, os arquitetos também precisam determinar quais são os requisitos funcionais, o que pode ser uma grande tarefa. Os requisitos podem ser recursos específicos inspirados pelo mercado. O software de aplicação normalmente controla a escolha de certos requisitos funcionais, determinando como o computador será usado. Se houver um grande conjunto de software para certa arquitetura de conjunto de instruções, o arquiteto poderá decidir que o novo computador deve implementar um dado conjunto de instruções. A presença de um grande mercado para determinada classe de aplicações pode encorajar os projetistas a incorporarem requisitos que tornariam o computador competitivo nesse mercado. Muitos desses requisitos e recursos são examinados em profundidade nos próximos capítulos.

Os arquitetos precisam estar conscientes das tendências importantes, tanto na tecnologia como na utilização dos computadores, já que elas afetam não somente os custos no futuro como também a longevidade de uma arquitetura.

1.4 TENDÊNCIAS NA TECNOLOGIA

Para ser bem-sucedida, uma arquitetura de conjunto de instruções precisa ser projetada para sobreviver às rápidas mudanças na tecnologia dos computadores. Afinal, uma nova arquitetura de conjunto de instruções bem-sucedida pode durar décadas — por exemplo, o núcleo do mainframe IBM está em uso há quase 50 anos. Um arquiteto precisa planejar visando às mudanças de tecnologia que possam aumentar o tempo de vida de um computador bem-sucedido.

Para planejar a evolução de um computador, o projetista precisa estar ciente das rápidas mudanças na tecnologia de implementação. Quatro dessas tecnologias, que mudam em ritmo notável, são fundamentais para as implementações modernas:

- *Tecnologia do circuito lógico integrado.* A densidade de transistores aumenta em cerca de 35% ao ano, quadruplicando em pouco mais de quatro anos. Os aumentos no

Requisitos funcionais	Recursos típicos exigidos ou admitidos
Área de aplicação	*Destino do computador*
Dispositivo móvel pessoal	Desempenho em tempo real para várias tarefas, incluindo desempenho interativo para gráficos, vídeo e áudio; eficiência energética (Caps. 2, 3, 4, 5; Apêndice A)
Desktop de uso geral	Desempenho balanceado para uma gama de tarefas, incluindo desempenho interativo para gráficos, vídeo e áudio (Caps. 2, 3, 4, 5; Apêndice A)
Servidores	Suporte para bancos de dados e processamento de transações; melhorias para confiabilidade e disponibilidade; suporte para escalabilidade (Caps. 2, 5; Apêndices A, D, F)
Clusters/computadores em escala wharehouse	Desempenho de throughput para muitas tarefas independentes; correção de erro de memória; proporcionalidade de energia (Caps. 2, 6; Apêndice F)
Computação embarcada	Normalmente exige suporte especial para gráficos ou vídeos (ou outra extensão específica da aplicação); limitações de potência e controle de potência podem ser exigidas (Caps. 2, 3, 5; Apêndices A, E)
Nível de compatibilidade de software	*Determina quantidade de software existente para o computador*
Na linguagem de programação	Mais flexível para o projetista; precisa de novo compilador (Caps. 3, 5; Apêndice A)
Código objeto ou binário compatível	A arquitetura do conjunto de instruções é completamente definida ¾ pouca flexibilidade ¾, mas nenhum investimento é necessário no software ou na portabilidade de programas (Apêndice A)
Requisitos do sistema operacional	*Recursos necessários para dar suporte ao sistema operacional escolhido (Cap. 2; Apêndice B)*
Tamanho do espaço de endereços	Recurso muito importante (Cap. 2); pode limitar as aplicações
Gerenciamento de memória	Exigido para o sistema operacional moderno; pode ser paginada ou segmentada (Cap. 2)
Proteção	Diferentes necessidades do sistema operacional e da aplicação: página versus segmento; máquinas virtuais (Cap. 2)
Padrões	*Certos padrões podem ser exigidos pelo mercado*
Ponto flutuante	Formato e aritmética: padrão IEEE 754 (Apêndice J), aritmética especial para gráficos ou processamento de sinal
Interfaces de E/S	Para dispositivos de E/S: Serial ATA, Serial Attach SCSI, PCI Express (Apêndices D, F)
Sistemas operacionais	UNIX, Windows, Linux, CISCO IOS
Redes	Suporte exigido para diferentes redes: Ethernet, Infiniband (Apêndice F)
Linguagens de programação	Linguagens (ANSI C, C++, Java, FORTRAN) afetam o conjunto de instruções (Apêndice A)

FIGURA 1.7 Resumo de alguns dos requisitos funcionais mais importantes com os quais um arquiteto se depara.
A coluna da esquerda descreve a classe de requisitos, enquanto a coluna da direita oferece exemplos específicos. A coluna da direita também contém referências a capítulos e apêndices que lidam com os requisitos específicos.

tamanho do die são menos previsíveis e mais lentos, variando de 10-20% por ano. O efeito combinado é o crescimento na contagem de transistores de um chip em cerca de 40-55% por ano, ou dobrando a cada 18-24 meses. Essa tendência é conhecida popularmente como *lei de Moore*. A velocidade do dispositivo aumenta mais lentamente, conforme mencionamos a seguir.

- *DRAM semicondutora* (memória dinâmica de acesso aleatório). Agora que a maioria dos chips DRAM é produzida principalmente em módulos DIMM, é mais difícil rastrear a capacidade do chip, já que os fabricantes de DRAM costumam oferecer produtos de diversas capacidades ao mesmo tempo, para combinar com a capacidade do DIMM. A capacidade por chip DRAM tem aumentado

Edição CA:AQA	Ano	Taxa de crescimento da DRAM (por ano)	Caracterização do impacto sobre a capacidade da DRAM
1	1990	60%	Quadruplica a cada 3 anos
2	1996	60%	Quadruplica a cada 3 anos
3	2003	40-60%	Quadriplica a cada 3-4 anos
4	2007	40%	Dobra a cada 2 anos
5	2011	25-40%	Dobra a cada 2-3 anos

FIGURA 1.8 **Mudança na taxa de melhoria na capacidade da DRAM ao longo do tempo.**
As duas primeiras edições chamaram essa taxa de regra geral de crescimento da DRAM, uma vez que havia sido bastante confiável desde 1977, com a DRAM de 16 kilobits, até 1996, com a DRAM de 64 megabits. Hoje, alguns questionam se a capacidade da DRAM pode melhorar em 5- 7 anos, devido a dificuldades em fabricar uma célula de DRAM cada vez mais tridimensional (Kim, 2005).

em cerca de 25-40% por ano, dobrando aproximadamente a cada 2-3 anos. Essa tecnologia é a base da memória principal e será discutida no Capítulo 2. Observe que a taxa de melhoria continuou a cair ao longo das edições deste livro, como mostra a Figura 1.8. Existe até mesmo uma preocupação: a taxa de crescimento vai parar no meio desta década devido à crescente dificuldade em produzir com eficiência células DRAM ainda menores (Kim, 2005)? O Capítulo 2 menciona diversas outras tecnologias que podem substituir o DRAM, se ele atingir o limite da capacidade.

- *Flash semicondutor* (*memória somente para leitura eletricamente apagável e programável*). Essa memória semicondutora não volátil é o dispositivo-padrão de armazenamento nos PMDs, e sua popularidade alavancou sua rápida taxa de crescimento em capacidade. Recentemente, a capacidade por chip Flash vem aumentando em cerca de 50-60% por ano, dobrando aproximadamente a cada dois anos. Em 2011, a memória Flash era 15-20 vezes mais barata por bit do que a DRAM. O Capítulo 2 descreve a memória Flash.
- *Tecnologia de disco magnético.* Antes de 1990, a densidade aumentava em cerca de 30% por ano, dobrando em três anos. Ela aumentou para 60% por ano depois disso e para 100% por ano em 1996. Desde 2004, caiu novamente para cerca de 40% por ano ou dobrou a cada três anos. Os discos são 15-25 vezes mais baratos por bit do que a Flash. Dada a taxa de crescimento reduzido da DRAM, hoje os discos são 300-500 vezes mais baratos por bit do que a DRAM. É a principal tecnologia para o armazenamento em servidores e em computadores em escala warehouse (vamos discutir essas tendências em detalhes no Apêndice D).
- *Tecnologia de rede.* O desempenho da rede depende do desempenho dos switches e do desempenho do sistema de transmissão (examinaremos as tendências em redes no Apêndice F).

Essas tecnologias que mudam rapidamente modelam o projeto de um computador que, com melhorias de velocidade e tecnologia, pode ter um tempo de vida de 3-5 anos. As principais tecnologias, como DRAM, Flash e disco, mudam o suficiente para que o projetista precise planejar essas alterações. Na realidade, em geral, os projetistas projetam para a próxima tecnologia sabendo que, quando um produto começar a ser entregue em volume, essa tecnologia pode ser a mais econômica ou apresentar vantagens de desempenho. Tradicionalmente, o custo tem diminuído aproximadamente na mesma taxa em que a densidade tem aumentado.

Embora a tecnologia melhore continuamente, o impacto dessas melhorias pode ocorrer em saltos discretos, à medida que um novo patamar para uma capacidade seja alcançado. Por exemplo, quando a tecnologia MOS atingiu um ponto, no início da década de 1980,

quando cerca de 25.000-50.000 transistores poderiam caber em um único chip, foi possível montar um microprocessador de único chip de 32 bits. Ao final da mesma década, as caches de primeiro nível puderam ser inseridos no mesmo chip. Eliminando os cruzamentos do processador dentro do chip e entre o processador e a cache, foi possível alcançar uma melhoria incrível no custo-desempenho e na potência-desempenho. Esse projeto era simplesmente inviável até que a tecnologia alcançasse determinado ponto. Com os microprocessadores multicore e número de cores aumentando a cada geração, mesmo os computadores servidores estão se dirigindo para ter um único chip para todos os processadores. Esses limites de tecnologia não são raros e possuem um impacto significativo sobre grande variedade de decisões de projeto.

Tendências de desempenho: largura de banda sobre latência

Como veremos na Seção 1.8, *largura de banda* ou *throughput* é a quantidade total de trabalho feito em determinado tempo, como megabytes por segundo, para uma transferência de disco. Ao contrário, *latência* ou *tempo de resposta* é o tempo entre o início e o término de um evento, como milissegundos, para um acesso ao disco. A Figura 1.9 representa a melhoria relativa na largura de banda e a latência para os marcos da tecnologia de microprocessadores, memória, redes e discos. A Figura 1.10 descreve os exemplos e os marcos com mais detalhes.

O desempenho é o principal diferenciador para microprocessadores e redes, de modo que eles têm visto os maiores ganhos: 10.000-20.000X em largura de banda e 30-80X em latência. A capacidade geralmente é mais importante do que o desempenho para memória e discos, de modo que a capacidade melhorou mais, embora seus avanços de largura de banda de 300-1.200X ainda sejam muito maiores do que seus ganhos em latência de 6-8X.

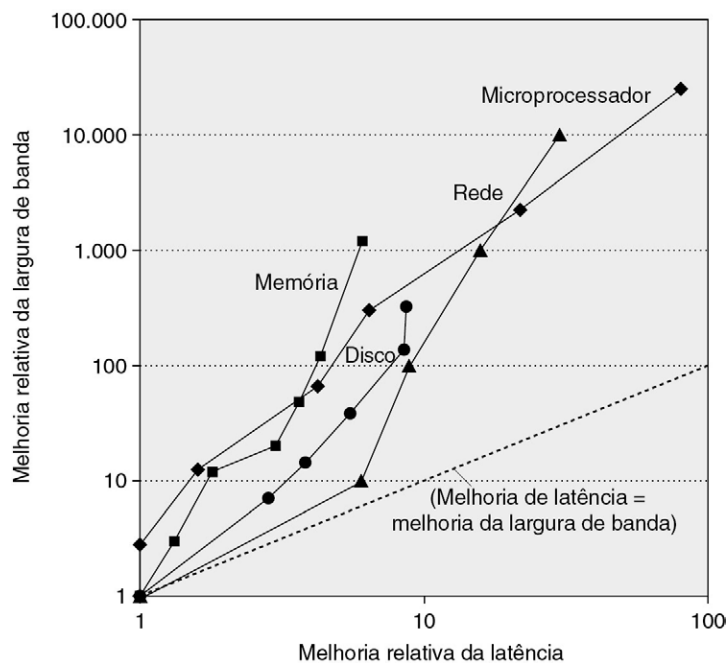



FIGURA 1.9 Representação simples dos marcos de largura de banda e latência da Figura 1.10 em relação ao primeiro marco.

Observe que a latência melhorou de 6X a 80X, enquanto a largura de banda melhorou cerca de 300X a 25.000X. Atualização de Patterson (2004).

Microprocessador	Barramento de endereços de 16 bits, microcodificado	Barramento de endereços de 32 bits, microcodificado	Pipeline em 5 estágios, caches I&D no chip, FPU	Barramento de 2 vias, superescalar, 64 bits	Superescalar de 3 vias fora de ordem	Superpipelined fora de ordem, cache 1.2 no chip	Multicore OOO 4 rotas cache L3 no chip, Turbo
Produto	Intel 80286	Intel 80386	Intel 80486	Intel Pentium	Intel Pentium Pro	Intel Pentium 4	Intel Core i7
Ano	1982	1985	1989	1993	1997	2001	2010
Tamanho do die (mm^2)	47	43	81	90	308	217	240
Transistores	134.000	275.000	1.200.000	3.100.000	5.500.000	42.000.000	1.170.000.000
Processadores/chip	1	1	1	1	1	1	4
Pinos	68	132	168	273	387	423	1.366
Latência (clocks)	6	5	5	5	10	22	14
Largura de banda (bits)	16	32	32	64	64	64	196
Taxa de clock (MHz)	12,5	16	25	66	200	1.500	3.333
Largura de banda (MIPS)	2	6	25	132	600	4.500	50.000
Latência (ns)	320	313	200	76	50	15	4
Módulo de memória	DRAM	DRAM em modo de página	DRAM rápida em modo de página	DRAM rápida em modo de página	DRAM síncrona	SDRAM com taxa de dados dupla	DDR3 SDRAM
Largura do módulo (bits)	16	16	32	64	64	64	64
Ano	1980	1983	1986	1993	1997	2000	2010
Mbits/chip DRAM	0,06	0,25	1	16	64	256	2048
Tamanho do die (mm^2)	35	45	70	130	170	204	50
Pinos/chip de DRAM	16	16	18	20	54	66	134
Largura de banda (MBit/s)	13	40	160	267	640	1.600	16.000
Latência (ns)	225	170	125	75	62	52	37
Rede local	Ethernet	Fast Ethernet	Gigabit Ethernet	10 Gigabit Ethernet	100 Gigabit Ethernet		
Padrão IEEE	802.3	803.3u	802.3ab	802.3ac	802.3ba		
Ano	1978	1995	1999	2003	2010		
Largura de banda (Mb/s)	10	100	1.000	10.000	100.000		
Latência (μs)	3.000	500	340	190	100		
Disco rígido	3.600 RPM	5.400 RPM	7.200 RPM	10.000 RPM	15.000 RPM	15.000 RPM	
Produto	CDC Wrenl 94145-36	Seagate ST41600	Seagate ST15150	Seagate ST39102	Seagate ST373453	Seagate ST3600057	
Ano	1983	1990	1994	1998	2003	2010	
Capacidade (GB)	0,03	1,4	4,3	9,1	73,4	600	
Form factor do disco	5,25"	5,25"	3,5"	3,5"	3,5"	3,5"	
Diâmetro da mídia	5,25"	5,25"	3,5"	3,0"	2,5"	2,5"	
Interface	ST-412	SCSI	SCSI	SCSI	SCSI	SAS	
Largura de banda (Mb/s)	0,6	4	9	24	86	204	
Latência (ms)	48,3	17,1	12,7	8,8	5,7	5,6	

FIGURA 1.10 **Marcos de desempenho por 25-40 anos para microprocessadores, memória, redes e discos.**
Os marcos do microprocessador são várias gerações de processadores IA-32, variando desde um 80286 microcodificado com barramento de 16 bits até um Core i7 multicor, com execução fora de ordem, superpipelined. Os marcos de módulo de memória vão da DRAM plana de 16 bits de largura até a DRAM síncrona versão 3 com taxa de dados dupla com 64 bits de largura. As redes Ethernet avançaram de 10 Mb/s até 100 Gb/s. Os marcos de disco são baseados na velocidade de rotação, melhorando de 3.600 RPM até 15.000 RPM. Cada caso é a largura de banda no melhor caso, e a latência é o tempo para uma operação simples, presumindo-se que não haja disputa. Atualização de Patterson (2004).

Claramente, a largura de banda ultrapassou a latência por essas tecnologias e provavelmente continuará dessa forma. Uma regra prática simples é que a largura de banda cresce ao menos pelo quadrado da melhoria na latência. Os projetistas de computadores devem levar isso em conta para o planejamento.

Escala de desempenho de transistores e fios

Os processos de circuito integrado são caracterizados pela *característica de tamanho*, que é o tamanho mínimo de um transistor ou de um fio na dimensão *x* ou *y*. Os tamanhos diminuíram de 10 μ em 1971 para 0,0032 μ em 2011; na verdade, trocamos as unidades, de modo que a produção em 2011 agora é referenciada como "32 nanômetros", e chips de 22 nanômetros estão a caminho. Como a contagem de transistores por milímetro quadrado de silício é determinada pela superfície de um transistor, a densidade de transistores quadruplica com uma diminuição linear no tamanho do recurso.

Porém, o aumento no desempenho do transistor é mais complexo. À medida que os tamanhos diminuem, os dispositivos encolhem quadruplicadamente nas dimensões horizontal e vertical. O encolhimento na dimensão vertical requer uma redução na voltagem de operação para manter a operação e a confiabilidade dos transistores correta. Essa combinação de fatores de escala leva a um inter-relacionamento entre o desempenho do transistor e a carecterística de tamanho do processo. Para uma primeira aproximação, o desempenho do transistor melhora linearmente com a diminuição de seu tamanho.

O fato de a contagem de transistores melhorar em quatro vezes, com uma melhoria linear no desempenho do transistor, é tanto o desafio quanto a oportunidade para a qual os arquitetos de computadores foram criados! Nos primeiros dias dos microprocessadores, a taxa de melhoria mais alta na densidade era usada para passar rapidamente de microprocessadores de 4 bits para 8 bits, para 16 bits, para 32 bits, para 64 bits. Mais recentemente, as melhorias de densidade admitiram a introdução de múltiplos microprocessadores por chip, unidades SIMD maiores, além de muitas das inovações em execução especulativa e em caches encontradas nos Capítulos 2, 3, 4 e 5.

Embora os transistores geralmente melhorem em desempenho com a diminuição do tamanho, os fios em um circuito integrado não melhoram. Em particular, o atraso de sinal em um fio aumenta na proporção com o produto de sua resistência e de sua capacitância. Naturalmente, à medida que o tamanho diminui, os fios ficam mais curtos, mas a resistência e a capacitância por tamanho unitário pioram. Esse relacionamento é complexo, pois tanto a resistência quanto a capacitância dependem de aspectos detalhados do processo, da geometria de um fio, da carga sobre um fio e até mesmo da adjacência com outras estruturas. Existem aperfeiçoamentos ocasionais no processo, como a introdução de cobre, que oferecem melhorias de uma única vez no atraso do fio.

Porém, em geral, o atraso do fio não melhora muito em comparação com o desempenho do transistor, criando desafios adicionais para o projetista. Nos últimos anos, o atraso do fio tornou-se uma limitação de projeto importante para grandes circuitos integrados e normalmente é mais crítico do que o atraso do chaveamento do transistor. Frações cada vez maiores de ciclo de clock têm sido consumidas pelo atraso de propagação dos sinais nos fios.

1.5 TENDÊNCIAS NA ALIMENTAÇÃO DOS CIRCUITOS INTEGRADOS

Hoje, a energia é o segundo maior desafio enfrentado pelo projetista de computadores para praticamente todas as classes de computador. Primeiramente, a alimentação precisa

ser trazida e distribuída pelo chip, e os microprocessadores modernos utilizam centenas de pinos e várias camadas de interconexão apenas para alimentação e terra. Além disso, a energia é dissipada como calor e precisa ser removida.

Potência e energia: uma perspectiva de sistema

Como um arquiteto de sistema ou um usuário devem pensar sobre o desempenho, a potência e a energia? Do ponto de vista de um projetista de sistema, existem três preocupações principais.

Em primeiro lugar, qual é a potência máxima que um processador pode exigir? Atender a essa demanda pode ser importante para garantir a operação correta. Por exemplo, se um processador tenta obter mais potência do que a fornecida por um sistema de alimentação (obtendo mais corrente do que o sistema pode fornecer), em geral o resultado é uma queda de tensão que pode fazer o dispositivo falhar. Processadores modernos podem variar muito em consumo de potência com altos picos de corrente. Portanto, eles fornecem métodos de indexação de tensão que permitem ao processador ficar mais lento e regular a tensão dentro de uma margem grande. Obviamente, fazer isso diminui o desempenho.

Em segundo lugar, qual é o consumo de potência sustentado? Essa métrica é amplamente chamada *projeto térmico de potência* (Thermal Design Power — TDP), uma vez que determina o requisito de resfriamento. O TDP não é nem potência de pico, em alguns momentos cerca de 1,5 vez maior, nem a potência média real que será consumida durante um dado cálculo, que provavelmente será ainda menor. Geralmente, uma fonte de alimentação típica é projetada para atender ou exceder o TDP. Não proporcionar resfriamento adequado vai permitir à temperatura de junção no processador exceder seu valor máximo, resultando em uma falha no dispositivo, possivelmente com danos permanentes. Os processadores modernos fornecem dois recursos para ajudar a gerenciar o calor, uma vez que a potência máxima (e, portanto, o calor e o aumento de temperatura) pode exceder, a longo prazo, a média especificada pela TDP. Primeiro, conforme a temperatura se aproxima do limite de temperatura de junção, os circuitos reduzem a taxa de clock, reduzindo também a potência. Se essa técnica não tiver sucesso, um segundo protetor de sobrecarga é ativado para desativar o chip.

O terceiro fator que os projetistas e usuários devem considerar é a energia e a eficiência energética. Lembre-se de que potência é simplesmente energia por unidade de tempo: 1 watt = 1 joule por segundo. Qual é a métrica correta para comparar processadores: energia ou potência? Em geral, a energia é sempre uma métrica melhor, porque está ligada a uma tarefa específica e ao tempo necessário para ela. Em particular, a energia para executar uma carga de trabalho é igual à potência média vezes o tempo de execução para a carga de trabalho.

Assim, se quisermos saber qual dentre dois processadores é mais eficiente para uma dada tarefa, devemos comparar o consumo de energia (não a potência) para realizá-la. Por exemplo, o processador A pode ter um consumo de potência médio 20% maior do que o processador B, mas se A executar a tarefa em apenas 70% do tempo necessário para B, seu consumo de energia será $1{,}2 \times 0{,}7 = 0{,}84$, que é obviamente melhor.

Pode-se argumentar que, em um grande servidor ou em uma nuvem, é suficiente considerar a potência média, uma vez que muitas vezes se supõe que a carga de trabalho seja infinita, mas isso é equivocado. Se nossa nuvem fosse ocupada por processadores B em vez de processadores A, faria menos trabalho pela mesma quantidade de energia gasta. Usar a energia para comparar as alternativas evita essa armadilha. Seja uma carga de trabalho fixa, uma nuvem warehouse-scale, seja um smartphone, comparar a energia será o modo correto de comparar alternativas de processador, já que tanto a conta de eletricidade para

a nuvem quanto o tempo de vida da bateria para o smartphone são determinados pela energia consumida.

Quando o consumo de potência é uma medida útil? O uso primário legítimo é como uma restrição; por exemplo, um chip pode ser limitado a 100 watts. Isso pode ser usado como métrica se a carga de trabalho for fixa, mas então é só uma variação da verdadeira métrica de energia por tarefa.

Energia e potência dentro de um microprocessador

Para os chips de CMOS, o consumo de energia dominante tradicional tem ocorrido no chaveamento de transistores, também chamada *energia dinâmica*. A energia exigida por transistor é proporcional ao produto da capacitância de carga do transistor ao quadrado da voltagem:

$$\text{Energia}_{\text{dinâmica}} \propto \text{Carga capacitiva} \times \text{Voltagem}^2$$

Essa equação é a energia de pulso da transição lógica de 0→1→0 ou 1→0→1. A energia de uma única transição (0→1 ou 1→0) é, então:

$$\text{Energia}_{\text{dinâmica}} \propto 1/2\,\text{Carga capacitiva} \times \text{Voltagem}^2$$

A potência necessária por transistor é somente o produto da energia de uma transição multiplicada pela frequência das transições:

$$\text{Potência}_{\text{dinâmica}} \propto 1/2\,\text{Carga capacitiva} \times \text{Voltagem}^2 \times \text{Frequência de chaveamento}$$

Para uma tarefa fixa, reduzir a taxa de clock reduz a potência, mas não a energia.

Obviamente, a potência dinâmica e a energia são muito reduzidas quando se reduz a voltagem, por isso as voltagens caíram de 5V para pouco menos de 1V em 20 anos. A carga capacitiva é uma função do número de transistores conectados a uma saída e à tecnologia, que determina a capacitância dos fios e transistores.

Exemplo Hoje, alguns microprocessadores são projetados para ter voltagem ajustável, de modo que uma redução de 15% na voltagem pode resultar em uma redução de 15% na frequência. Qual seria o impacto sobre a energia dinâmica e a potência dinâmica?

Resposta Como a capacitância é inalterada, a resposta para a energia é a razão das voltagens, uma vez que a capacitância não muda:

$$\frac{\text{Energia}_{\text{nova}}}{\text{Energia}_{\text{velha}}} = 0{,}72 \times \frac{(\text{Voltagem} \times 0{,}85)^2}{\text{Voltagem}^2} = 0{,}85^2 = 0{,}72$$

reduzindo assim a potência para cerca de 72% da original. Para a potência, adicionamos a taxa das frequências

$$\frac{\text{Energia}_{\text{nova}}}{\text{Energia}_{\text{velha}}} = 0{,}72 \times \frac{(\text{Frequência de chaveamento} \times 0{,}85)}{\text{Frequência de chaveamento}} = 0{,}61$$

reduzindo a potência para cerca de 61% do original.

Ao passarmos de um processo para outro, o aumento no número de transistores chaveados e a frequência com que eles chaveiam dominam a diminuição na capacitância de carga e voltagem, levando a um crescimento geral no consumo de potência e energia. Os primeiros microprocessadores consumiam menos de 1 watt, e os primeiros microprocessadores de 32 bits (como o Intel 80386) usavam cerca de 2 watts, enquanto um Intel Core i7 de 3,3 GHz consome 130 watts. Visto que esse calor precisa ser dissipado de um chip com cerca de 1,5 cm em um lado, estamos alcançando os limites do que pode ser resfriado pelo ar.

Dada a equação anterior, você poderia esperar que o crescimento da frequência de clock diminuísse se não pudéssemos reduzir a voltagem ou aumentar a potência por chip. A Figura 1.11 mostra que esse é, de fato, o caso desde 2003, mesmo para os microprocessadores que tiveram os melhores desempenhos a cada ano. Observe que esse período de taxas constantes de clock corresponde ao período de baixa melhoria de desempenho na Figura 1.1.

Distribuir a potência, retirar o calor e impedir pontos quentes têm tornado-se desafios cada vez mais difíceis. A potência agora é a principal limitação para o uso de transistores; no passado, era a área bruta do silício. Portanto, os microprocessadores modernos oferecem muitas técnicas para tentar melhorar a eficiência energética, apesar das taxas de clock e tensões de alimentação constantes:

1. *Não fazer nada bem*. A maioria dos microprocessadores de hoje desliga o clock de módulos inativos para economizar energia e potência dinâmica. Por exemplo, se nenhuma instrução de ponto flutuante estiver sendo executada, o clock da unidade de ponto flutuante será desativado. Se alguns núcleos estiverem inativos, seus clocks serão interrompidos.
2. *Escalamento dinâmico de voltagem-frequência* (*Dynamic Voltage-Frequency Scaling — DVFS*). A segunda técnica vem diretamente das fórmulas anteriores. Dispositivos pessoais móveis, laptops e, até mesmo, servidores têm períodos de baixa atividade, em que não há necessidade de operar em frequências de clock e voltagens mais elevadas. Os microprocessadores modernos costumam oferecer algumas frequências

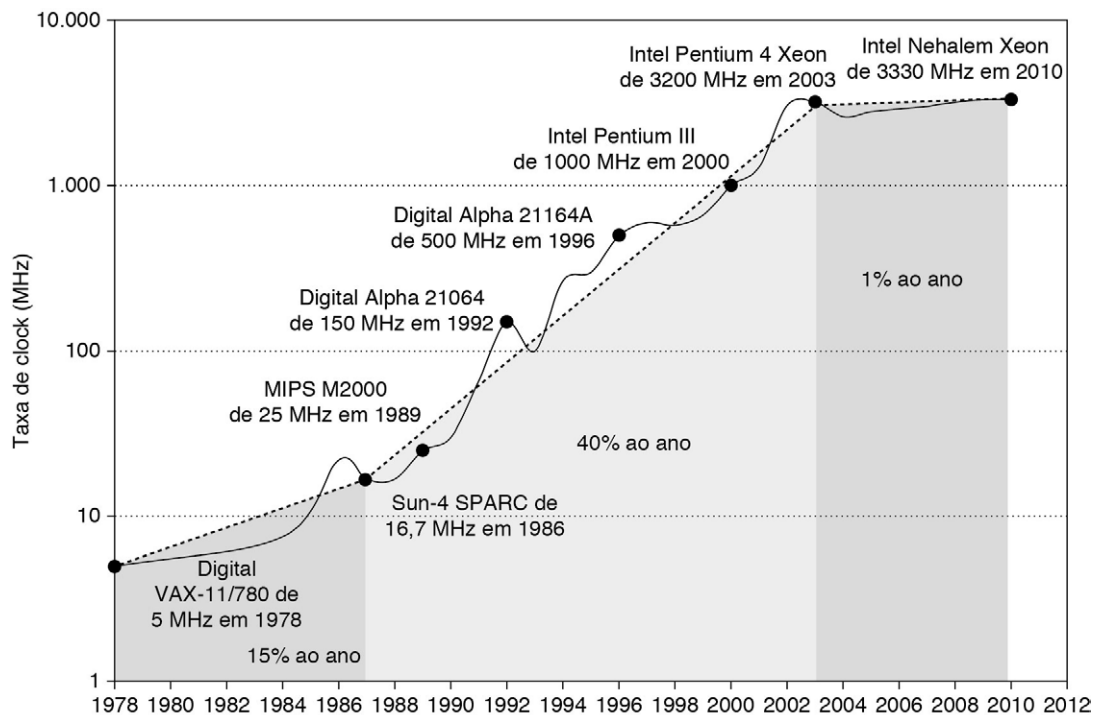



FIGURA 1.11 **Crescimento na taxa de clock dos microprocessadores na Figura 1.1.**
Entre 1978 e 1986, a taxa de clock aumentou menos de 15% por ano, enquanto o desempenho aumentou em 25% por ano. Durante o "período de renascimento" de 52% de melhoria de desempenho por ano entre 1986 e 2003, as taxas de clock aumentaram em quase 40% por ano. Desde então, a taxa de clock tem sido praticamente a mesma, crescendo menos de 1% por ano, enquanto o desempenho de processador único melhorou menos de 22% por ano.

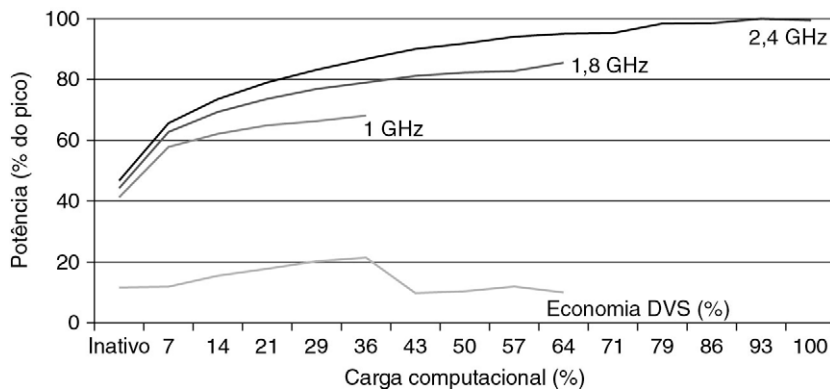



FIGURA 1.12 Economias de energia para um servidor usando um microprocessador AMD Opteron, 8 GB de DRAM, e um disco ATA.
A 1,8 GHz, o servidor só pode lidar até dois terços da carga de trabalho sem causar violações de nível de serviço, e a 1,0 GHz ele só pode lidar com a segurança de um terço da carga de trabalho (Figura 5.11, em Barroso e Hölzle, 2009).

de clock e voltagens que usam menor potência e energia. A Figura 1.12 mostra as economias potenciais de potência através de DVFS para um servidor, conforme a carga de trabalho diminui para três diferentes taxas de clock: 2,4 GHz, 1,8 GHz e 1 GHz. A economia geral de potência no servidor é de cerca de 10-15% para cada um dos dois passos.

3. *Projeto para um caso típico.* Dado que os PMDs e laptops muitas vezes estão inativos, a memória e o armazenamento oferecem modos de baixa potência para poupar energia. Por exemplo, DRAMs têm uma série de modos de potência cada vez menores para aumentar a vida da bateria em PMDs e laptops, e há propostas de discos que têm um modo de girar a taxas menores quando inativos, para poupar energia. Infelizmente, você não pode acessar DRAMs ou discos nesses modos, então deve retornar a um modo totalmente ativo para ler ou gravar, não importa quão baixa seja a taxa de acesso. Como mencionado, os microprocessadores para PCs, ao contrário, foram projetados para um caso mais típico de uso pesado a altas temperaturas de operação, dependendo dos sensores de temperatura no chip para detectar quando a atividade deve ser automaticamente reduzida para evitar sobreaquecimento. Essa "redução de velocidade de emergência" permite aos fabricantes projetar para um caso mais típico e, então, depender desse mecanismo de segurança se alguém realmente executar programas que consumam muito mais potência do que é típico.
4. *Overclocking*. Em 2008, a Intel começou a oferecer o *modo Turbo*, em que o chip decide que é seguro rodar a uma taxa maior de clock por um curto período, possivelmente em alguns poucos núcleos, até que a temperatura comece a subir. Por exemplo, o Core i7 de 3,3 GHz pode rodar em explosões curtas a 3,6 GHz. De fato, todos os microprocessadores de maior desempenho a cada ano desde 2008, indicados na Figura 1.1, ofereceram overclocking temporário de cerca de 10% acima da taxa de clock nominal. Para código de thread único, esses microprocessadores podem desligar todos os núcleos, com exceção de um, e rodá-lo a uma taxa de clock ainda maior. Observe que, enquanto o sistema operacional pode desligar o modo Turbo, não há notificação, uma vez que ele seja habilitado. Assim, os programadores podem se surpreender ao ver que seus programas variam em desempenho devido à temperatura ambiente!

Embora a potência dinâmica seja a principal fonte de dissipação de potência na CMOS, a potência estática está se tornando uma questão importante, pois a corrente de fuga flui até mesmo quando um transistor está desligado:

$$\text{Potência}_{\text{estática}} \propto \text{Corrente}_{\text{estática}} \times \text{Voltagem}$$

ou seja, a potência estática é proporcional ao número de dispositivos.

Assim, aumentar o número de transistores aumenta a potência, mesmo que eles estejam inativos, e a corrente de fuga aumenta em processadores com transistores de menor tamanho. Como resultado, sistemas com muito pouca potência ainda estão passando a voltagem para módulos inativos, a fim de controlar a perda decorrente da fuga. Em 2011, a meta para a fuga era de 25% do consumo total de energia, mas a fuga nos projetos de alto desempenho muitas vezes ultrapassou bastante esse objetivo. A fuga pode ser de até 50% em tais chips, em parte por causa dos maiores caches de SRAM, que precisam de potência para manter os valores de armazenamento (o "S" em SRAM é de estático — *static*). A única esperança de impedir a fuga é desligar a alimentação de subconjuntos dos chips.

Por fim, como o processador é só uma parte do custo total de energia de um sistema, pode fazer sentido usar um que seja mais rápido e menos eficiente em termos de energia para permitir ao restante do sistema entrar em modo *sleep*. Essa estratégia é conhecida como *race-to-halt.*

A importância da potência e da energia aumentou o cuidado na avaliação sobre a eficiência de uma inovação, então agora a avaliação primária inclui as tarefas por joule ou desempenho por watt, ao contrário do desempenho por mm^2 de silício. Essa nova métrica afeta as abordagens do paralelismo, como veremos nos Capítulos 4 e 5.

1.6 TENDÊNCIAS NO CUSTO

Embora existam projetos de computador nos quais os custos costumam ser menos importantes — especificamente, supercomputadores —, projetos sensíveis ao custo tornam-se primordiais. Na realidade, nos últimos 30 anos, o uso de melhorias tecnológicas para reduzir o custo, além de aumentar o desempenho, tem sido um tema importante no setor de computação.

Os livros-texto normalmente ignoram a parte "custo" do par custo-desempenho, porque os custos mudam, tornando os livros desatualizados e porque essas questões são sutis, diferindo entre os segmentos do setor. Mesmo assim, ter compreensão do custo e de seus fatores é essencial para os projetistas tomarem decisões inteligentes quanto a um novo recurso ser ou não incluído nos projetos em que o custo é importante (imagine os arquitetos projetando prédios sem qualquer informação sobre os custos das vigas de aço e do concreto!).

Esta seção trata dos principais fatores que influenciam o custo de um computador e o modo como esses fatores estão mudando com o tempo.

O impacto do tempo, volume e *commodities*

O custo de um componente de computador manufaturado diminui com o tempo, mesmo sem que haja grandes melhorias na tecnologia de implementação básica. O princípio básico que faz os custos caírem é a *curva de aprendizado* — os custos de manufatura diminuem com o tempo. A própria curva de aprendizado é mais bem medida pela mudança no *rendimento* — a porcentagem dos dispositivos manufaturados que sobrevivem ao procedimento de teste. Seja um chip, uma placa, seja um sistema, os projetos que têm o dobro de rendimento terão a metade do custo.

Entender como a curva de aprendizado melhora o rendimento é fundamental para proteger os custos da vida de um produto. Um exemplo disso é que, a longo prazo, o preço por megabyte da DRAM tem caído. Como as DRAMs costumam ter seu preço relacionado com o custo — com exceção dos períodos de escassez ou de oferta em demasia —, o preço e o custo da DRAM andam lado a lado.

Os preços de microprocessadores também caem com o tempo, mas, por serem menos padronizados que as DRAMs, o relacionamento entre preço e custo é mais complexo. Em um período de competitividade significativa, o preço costuma acompanhar o custo mais de perto, embora seja provável que os vendedores de microprocessador quase nunca tenham perdas.

O volume é o segundo fator importante na determinação do custo. Volumes cada vez maiores afetam o custo de várias maneiras. Em primeiro lugar, eles reduzem o tempo necessário para diminuir a curva de aprendizado, que é parcialmente proporcional ao número de sistemas (ou chips) manufaturados. Em segundo lugar, o volume diminui o custo, pois aumenta a eficiência de compras e manufatura. Alguns projetistas estimaram que o custo diminui cerca de 10% para cada duplicação do volume. Além do mais, o volume diminui a quantidade de custo de desenvolvimento que precisa ser amortizada por computador, permitindo que os preços de custo e de venda sejam mais próximos.

Commodities são produtos essencialmente idênticos vendidos por vários fornecedores em grandes volumes. Quase todos os produtos ofertados nas prateleiras de supermercados são *commodities*, assim como DRAMs, memória Flash, discos, monitores e teclados comuns. Nos últimos 25 anos, grande parte da ponta inferior do negócio de computador tornou-se um negócio de *commodity*, focalizando a montagem de computadores desktop e laptops que rodam o Microsoft Windows.

Como muitos fornecedores entregam produtos quase idênticos, isso é altamente competitivo. É natural que essa competição diminua a distância entre preço de custo e preço de venda, mas que também aumente o custo. As reduções ocorrem porque um mercado de *commodity* possui volume e clara definição de produto, de modo que vários fornecedores podem competir pela montagem dos componentes para o produto. Como resultado, o custo geral desse produto é mais baixo, devido à competição entre os fornecedores dos componentes e à eficiência de volume que eles podem conseguir. Isso fez com que o negócio de computador, de produtos finais, fosse capaz de alcançar melhor preço-desempenho do que outros setores e resultou em maior crescimento, embora com lucros limitados (como é comum em qualquer negócio de *commodity*).

Custo de um circuito integrado

Por que um livro sobre arquitetura de computadores teria uma seção sobre custos de circuito integrado? Em um mercado de computadores cada vez mais competitivo, no qual partes-padrão — discos, memória Flash, DRAMs etc. — estão tornando-se parte significativa do custo de qualquer sistema, os custos de circuito integrado tornam-se uma parte maior do custo que varia entre os computadores, especialmente na parte de alto volume do mercado, sensível ao custo. De fato, com a dependência cada vez maior dos dispositivos pessoais móveis em relação a *sistemas em um chip* (Systems On a Chip — SOC) completos, o custo dos circuitos integrados representa grande parte do custo do PMD. Assim, os projetistas de computadores precisam entender os custos dos chips para entender os custos dos computadores atuais.

Embora os custos dos circuitos integrados tenham caído exponencialmente, o processo básico de manufatura do silício não mudou: um *wafer* ainda é testado e cortado em *dies*,

que são encapsulados (Figs. 1.13, 1.14 e 1.15). Assim, o custo de um circuito integrado finalizado é:

$$\text{Custo do c.i.} = \frac{\text{Custo do die} + \text{Custo de testar o die} + \text{Custo de encapsulamento e teste final}}{\text{Rendimento de teste final}}$$

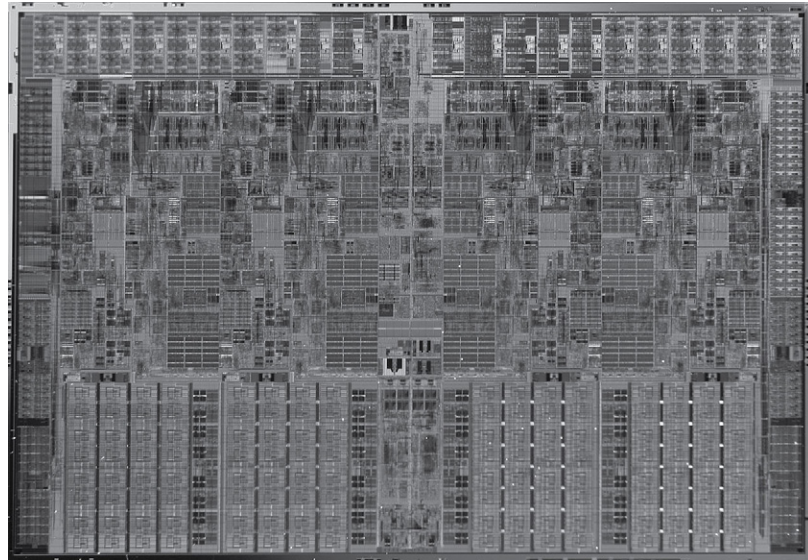



FIGURA 1.13 **Fotografia de um die de microprocessador Intel Core i7, que será avaliado nos Capítulos 2 a 5.** As dimensões são 18,9 mm por 13,6 mm (257 mm^2) em um processo de 45 nm (cortesia da Intel).

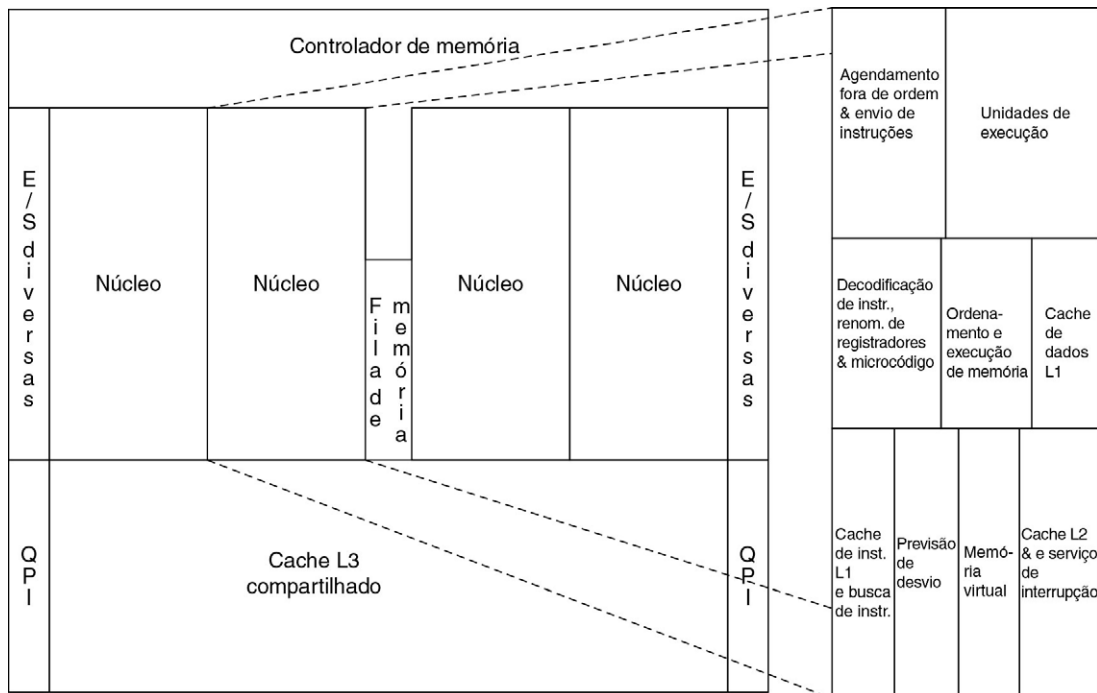



FIGURA 1.14 **Diagrama do die do Core i7 na Figura 1.13, à esquerda, com close-up do diagrama do segundo núcleo, à direita.**

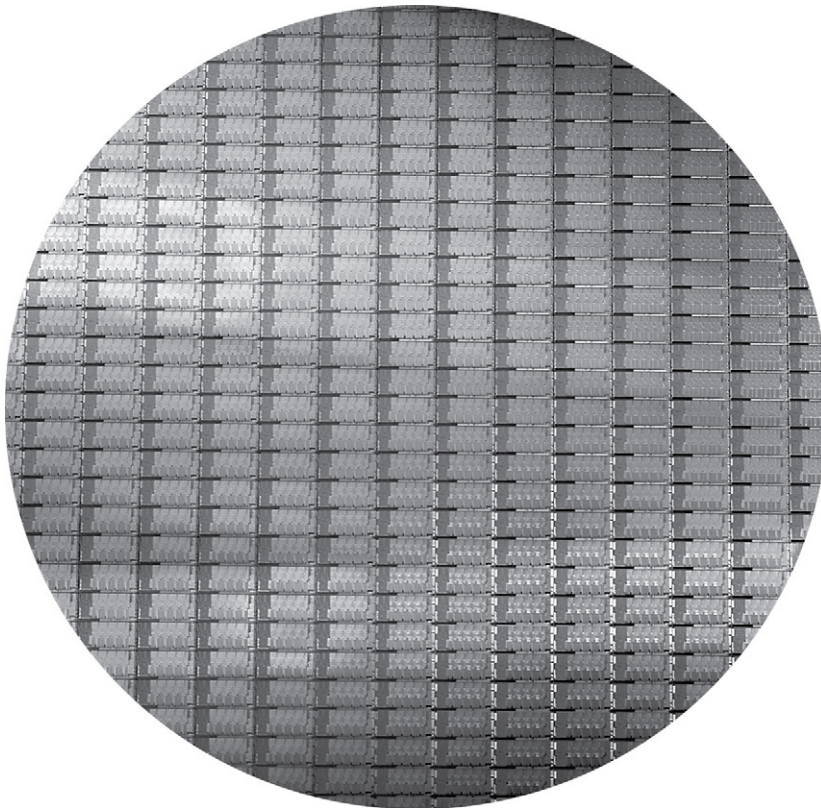



FIGURA 1.15 **Esse wafer de 300 mm contém 280 dies Sandy Dridge, cada um com 20,7 por 10,5 mm em um processo de 32 nm.**
(O Sandy Bridge é o sucessor da Intel para o Nehalem usado no Core i7.) Com 216 mm^2, a fórmula para dies por wafer estima 282 *(cortesia da Intel).*

Nesta seção, focalizamos o custo dos dies, resumindo os principais problemas referentes ao teste e ao encapsulamento no final.

Aprender a prever o número de chips bons por wafer exige aprender primeiro quantos dies cabem em um wafer e, depois, como prever a porcentagem deles que funcionará. A partir disso, é simples prever o custo:

$$\text{Custo do die} = \frac{\text{Custo do wafer}}{\text{Dies por wafer} \times \text{Rendimento do die}}$$

O recurso mais interessante desse primeiro termo da equação de custo do chip é sua sensibilidade ao tamanho do die, como veremos a seguir.

O número de dies por wafer é aproximadamente a área do wafer dividida pela área do die. Ela pode ser estimada com mais precisão por

$$\text{Dies por wafer} = \frac{\pi \times (\text{Diâmetro do wafer} / 2)^2}{\text{Área do die}} - \frac{\pi \times \text{Diâmetro do wafer}}{\sqrt{2 \times \text{Área do die}}}$$

O primeiro termo é a razão da área do wafer (πr^2) pela área do die. O segundo compensa o problema do "encaixe quadrado em um furo redondo" — dies retangulares perto da periferia de wafers redondos. A divisão da circunferência (πd) pela diagonal de um die quadrado é aproximadamente o número de dies ao longo da borda.

Exemplo Encontre o número de dies por wafer de 300 mm (30 cm) para um die que possui 1,5 cm em um lado e para um die que possui 1,0 cm em outro.

Resposta Quando a área do die é 2,25 cm^2:

$$\text{Dies por wafer} = \frac{\pi \times (30/2)^2}{2,25} - \frac{\pi \times 30}{\sqrt{2 \times 2,25}} = \frac{706,9}{2,25} = \frac{94,2}{2,12} = 270$$

Uma vez que a área do die maior é 2,25 vezes maior, há aproximadamente 2,25 vezes dies menores por wafer:

$$\text{Dies por wafer} = \frac{\pi \times (30/2)^2}{1,00} - \frac{\pi \times 30}{\sqrt{2 \times 1,00}} = \frac{706,9}{1,00} = \frac{94,2}{1,41} = 640$$

Porém, essa fórmula só dá o número máximo de dies por wafer. A questão decisiva é: qual é a fração de dies bons em um wafer, ou seja, o *rendimento de dies*? Um modelo simples de rendimento de circuito integrado que considera que os defeitos são distribuídos aleatoriamente sobre o wafer e esse rendimento é inversamente proporcional à complexidade do processo de fabricação, leva ao seguinte:

$$\text{Rendimento do die} = \text{Rendimento do wafer} \times 1/(1 + \text{Defeitos por unidade de área} \times \text{Área do die})^N$$

Essa fórmula de Bose-Einstein é um modelo empírico desenvolvido pela observação do rendimento de muitas linhas de fabricação (Sydow, 2006). O *rendimento do wafer* considera os wafers que são completamente defeituosos e, por isso, não precisam ser testados. Para simplificar, vamos simplesmente considerar que o rendimento do wafer seja de 100%. Os defeitos por unidade de área são uma medida dos defeitos de fabricação aleatórios que ocorrem. Em 2010, esse valor normalmente é de 0,1-0,3 defeito por centímetro quadrado, ou 0,016-0,057 defeito por centímetro quadrado, para um processo de 40 nm, pois isso depende da maturidade do processo (lembre-se da curva de aprendizado mencionada). Por fim, N é um parâmetro chamado *fator de complexidade da fabricação*. Para processos de 40 nm em 2010, N variou de 11,5-15,5.

Exemplo Encontre o rendimento para os dies com 1,5 cm de um lado e 1,0 cm de outro, considerando uma densidade de defeito de 0,031 por cm^2 e N de 13,5.

Resposta As áreas totais de die são 2,25 cm^2 e 1,00 cm^2. Para um die maior, o rendimento é:

$$\text{Rendimento do die} = 1/(1 + 0,031 \times 2,25)^{13,5} = 0,40$$

Para o die menor, ele é o rendimento do die:

$$\text{Rendimento do die} = 1/(1 + 0,031 \times 1,00)^{13,5} = 0,66$$

Ou seja, menos da metade de todo o die grande é bom, porém mais de dois terços do die pequeno são bons.

O resultado é o número de dies bons por wafer, que vem da multiplicação dos dies por wafer pelo rendimento do die usado para incorporar os impactos dos defeitos. Os exemplos anteriores preveem cerca de 109 dies bons de 2,25 cm^2 e 424 dies bons de 1,00 cm^2 no wafer de 300 mm. Muitos microprocessadores se encontram entre esses dois tamanhos. Os processadores embarcados de 32 bits de nível inferior às vezes possuem até 0,10 cm^2,

e os processadores usados para controle embarcado (em impressoras, automóveis etc.) às vezes têm menos de 0,04 cm^2.

Devido às consideráveis pressões de preço sobre os produtos de *commodity*, como DRAM e SRAM, os projetistas incluíram a redundância como um meio de aumentar o rendimento. Por vários anos, as DRAMs regularmente incluíram algumas células de memória redundantes, de modo que certo número de falhas possa ser acomodado. Os projetistas têm usado técnicas semelhantes, tanto em SRAMs padrão quanto em grandes arrays de SRAM, usados para caches dentro dos microprocessadores. É óbvio que a presença de entradas redundantes pode ser usada para aumentar significativamente o rendimento.

O processamento de um wafer de 300 mm (12 polegadas) de diâmetro em tecnologia de ponta custava US$ 5.000-6.000 em 2010. Considerando um custo de wafer processado de US$ 5.500, o custo do die de 1,00 cm^2 seria em torno de US$ 13, mas o custo por die de 2,25 cm^2 seria cerca de US$ 51, quase quatro vezes o custo para um die com pouco mais que o dobro do tamanho.

Por que um projetista de computador precisa se lembrar dos custos do chip? O processo de manufatura dita o custo e o rendimento do wafer, e os defeitos por unidade de área, de modo que o único controle do projetista é a área do die. Na prática, como o número de defeitos por unidade de área é pequeno, o número de dies bons por wafer e, portanto, o custo por die crescem rapidamente, conforme o quadrado da área do die. O projetista de computador afeta o tamanho do die e, portanto, o custo, tanto pelas funções incluídas ou excluídas quanto pelo número de pinos de E/S.

Antes que tenhamos uma parte pronta para uso em um computador, os dies precisam ser testados (para separar os bons dies dos ruins), encapsulados e testados novamente após o encapsulamento. Esses passos aumentam consideravelmente os custos.

Essa análise focalizou os custos variáveis da produção de um die funcional, que é apropriado para circuitos integrados de alto volume. Porém, existe uma parte muito importante do custo fixo que pode afetar significativamente o custo de um circuito integrado para baixos volumes (menos de um milhão de partes), o custo de um conjunto de máscaras. Cada etapa do processo de circuito integrado requer uma máscara separada. Assim, para os modernos processos de fabricação de alta densidade, com quatro a seis camadas de metal, os custos por máscara ultrapassam US$ 1 milhão. Obviamente, esse grande custo fixo afeta o custo das rodadas de prototipagem e depuração, e — para produção em baixo volume — pode ser uma parte significativa do custo de produção. Como os custos por máscara provavelmente continuarão a aumentar, os projetistas podem incorporar a lógica reconfigurável para melhorar a flexibilidade de uma parte ou decidir usar gate-arrays (que possuem número menor de níveis de máscara de customização) e, assim, reduzir as implicações de custo das máscaras.

Custo *versus* preço

Com os computadores se tornando *commodities*, a margem entre o custo para a manufatura de um produto e o preço pelo qual o produto é vendido tem diminuído. Essa margem considera a pesquisa e desenvolvimento (P&D), o marketing, as vendas, a manutenção do equipamento de manufatura, o aluguel do prédio, o custo do financiamento, os lucros pré-taxados e os impostos de uma empresa. Muitos engenheiros ficam surpresos ao descobrir que a maioria das empresas gasta apenas de 4% (no negócio de PC *commodity*) a 12% (no negócio de servidor de alto nível) de sua receita em P&D, que inclui toda a engenharia.

Custo de fabricação *versus* custo de operação

Nas primeiras quatro edições deste livro, *custo* queria dizer o valor gasto para construir um computador e *preço* significava a quantia para comprar um computador. Com o advento de computadores em escala warehouse, que contêm dezenas de milhares de servidores, o custo de operar os computadores é significativo em adição ao custo de compra.

Como o Capítulo 6 mostra, o preço de compra amortizado dos servidores e redes corresponde a um pouco mais de 60% do custo mensal para operar um computador em escala warehouse, supondo um tempo de vista curto do equipamento de TI de 3-4 anos. Cerca de 30% dos custos operacionais mensais têm relação com o uso de energia e a infraestrutura amortizada para distribuí-la e resfriar o equipamento de TI, apesar de essa infraestrutura ser amortizada ao longo de 10 anos. Assim, para reduzir os custos em um computador em escala de warehouse, os arquitetos de computadores precisam usar a energia com eficiência.

1.7 DEPENDÊNCIA

Historicamente, os circuitos integrados sempre foram um dos componentes mais confiáveis de um computador. Embora seus pinos fossem vulneráveis e pudesse haver falhas nos canais de comunicação, a taxa de erro dentro do chip era muito baixa. Mas essa sabedoria convencional está mudando à medida que chegamos a tamanhos de 32 nm e ainda menores: falhas transientes e permanentes se tornarão mais comuns, de modo que os arquitetos precisarão projetar sistemas para lidar com esses desafios. Esta seção apresenta um rápido panorama dessas questões de dependência, deixando a definição oficial dos termos e das técnicas para a Seção D.3 do Apêndice D.

Os computadores são projetados e construídos em diferentes camadas de abstração. Podemos descer recursivamente por um computador, vendo os componentes se ampliarem para subsistemas completos até nos depararmos com os transistores individuais. Embora algumas falhas, como a falta de energia, sejam generalizadas, muitas podem ser limitadas a um único componente em um módulo. Assim, a falha pronunciada de um módulo em um nível pode ser considerada meramente um erro de componente em um módulo de nível superior. Essa distinção é útil na tentativa de encontrar maneiras de montar computadores confiáveis.

Uma questão difícil é decidir quando um sistema está operando corretamente. Esse ponto filosófico tornou-se concreto com a popularidade dos serviços de internet. Os provedores de infraestrutura começaram a oferecer *Service Level Agreements* (SLA) ou *Service Level Objectives* (SLO) para garantir que seu serviço de rede ou energia fosse confiável. Por exemplo, eles pagariam ao cliente uma multa se não cumprissem um acordo por mais de algumas horas por mês. Assim, um SLA poderia ser usado para decidir se o sistema estava ativo ou inativo.

Os sistemas alternam dois *status* de serviço com relação a um SLA:

1. *Realização do serviço*, em que o serviço é entregue conforme o que foi especificado.
2. *Interrupção de serviço*, em que o serviço entregue é diferente do SLA.

As transições entre esses dois *status* são causadas por *falhas* (do *status* 1 para o 2) ou *restaurações* (do *status* 2 para o 1). Quantificar essas transições leva às duas principais medidas de dependência:

- *Confiabilidade do módulo* é uma medida da realização contínua do serviço (ou, de forma equivalente, do tempo para a falha) de um instante inicial de referência. Logo, o *tempo médio para a falha* (Mean Time To Failure — MTTF) é uma medida

de confiabilidade. O recíproco do MTTF é uma taxa de falhas, geralmente informada como falhas por bilhão de horas de operação ou *falhas em tempo* (Failures In Time — FIT). Assim, um MTTF de 1.000.000 de horas é igual a $10^9/10^6$ ou 1.000 FIT. A interrupção do serviço é medida como *tempo médio para o reparo* (Mean Time To Repair — MTTR). O *tempo médio entre as falhas* (Mean Time Between Failures — MTBF) é simplesmente a soma MTTF + MTTR. Embora o MTBF seja bastante usado, normalmente o MTTF é o termo mais apropriado. Se uma coleção de módulos tiver tempos de vida distribuídos exponencialmente — significando que a idade de um módulo não é importante na probabilidade de falha —, a taxa de falha geral do conjunto é a soma das taxas de falha dos módulos.

- *Disponibilidade do módulo* é uma medida da realização do serviço com relação à alternância de dois *status* de realização e interrupção. Para sistemas não redundantes com reparo, a disponibilidade do módulo é

$$\text{Disponibilidade do módulo} = \frac{\text{MTTF}}{(\text{MTTF} + \text{MTTR})}$$

Observe que agora a confiabilidade e a disponibilidade são medições quantificáveis, em vez de sinônimos de dependência. A partir dessas definições, podemos estimar a confiabilidade de um sistema quantitativamente se fizermos algumas suposições sobre a confiabilidade dos componentes e se essas falhas forem independentes.

Exemplo Considere um subsistema de disco com os seguintes componentes e MTTF:

- 10 discos, cada qual classificado em 1.000.000 horas de MTTF
- 1 controladora SCSI, 500.000 horas de MTTF
- 1 fonte de alimentação, 200.000 horas de MTTF
- 1 ventilador, 200.000 horas de MTTF
- 1 cabo SCSI, 1.000.000 de horas de MTTF

Usando as suposições simplificadas de que os tempos de vida são distribuídos exponencialmente e de que as falhas são independentes, calcule o MTTF do sistema como um todo.

Resposta A soma das taxas de falha é:

$$\begin{aligned}\text{Taxa de falha}_{\text{sistema}} &= 10 \times \frac{1}{1.000.000} + \frac{1}{500.000} + \frac{1}{200.000} + \frac{1}{200.000} + \frac{1}{1.000.000} \\ &= \frac{10+2+5+5+1}{1.000.000\ \textit{horas}} + \frac{23}{1.000.000} + \frac{23.000}{1.000.000.000\ \text{horas}}\end{aligned}$$

ou 23.000 FIT. O MTTF para o sistema é exatamente o inverso da taxa de falha:

$$\text{MTTF}_{\text{sistema}} = \frac{1}{\text{Taxa de falha}_{\text{sistema}}} = \frac{1.000.000.000\ \text{horas}}{23.000} = 43.500\ \text{horas}$$

ou pouco menos de cinco anos.

A principal maneira de lidar com a falha é a redundância, seja em tempo (repita a operação para ver se ainda está com erro), seja em recursos (tenha outros componentes para utilizar no lugar daquele que falhou). Quando o componente é substituído e o sistema totalmente reparado, a dependência do sistema é considerada tão boa quanto nova. Vamos quantificar os benefícios da redundância com um exemplo.

Exemplo Os subsistemas de disco normalmente possuem fontes de alimentação redundantes para melhorar a sua pendência. Usando os componentes e MTTFs do exemplo anterior, calcule a confiabilidade de uma fonte de alimentação redundante. Considere que uma fonte de alimentação é suficiente para alimentar o subsistema de disco e que estamos incluindo uma fonte de alimentação redundante.

Resposta Precisamos de uma fórmula para mostrar o que esperar quando podemos tolerar uma falha e ainda oferecer serviço. Para simplificar os cálculos, consideramos que os tempos de vida dos componentes são distribuídos exponencialmente e que não existe dependência entre as falhas de componente. O MTTF para nossas fontes de alimentação redundantes é o tempo médio até que uma fonte de alimentação falhe dividido pela chance de que a outra falhará antes que a primeira seja substituída. Assim, se a possibilidade de uma segunda falha antes do reparo for pequena, o MTTF do par será grande. Como temos duas fontes de alimentação e falhas independentes, o tempo médio até que um disco falhe é $MTTF_{\text{fonte de alimentação}}/2$. Uma boa aproximação da probabilidade de uma segunda falha é MTTR sobre o tempo médio até que a outra fonte de alimentação falhe. Logo, uma aproximação razoável para um par redundante de fontes de alimentação é

$$MTTF_{\text{par de fontes de alimentação}} = \frac{MTTF_{\text{fonte de alimentação}} / 2}{\frac{MTTR_{\text{fonte de alimentação}}}{MTTF_{\text{fonte de alimentação}}}} = \frac{MTTF^2_{\text{fonte de alimentação}} / 2}{MTTR_{\text{fonte de alimentação}}} = \frac{MTTF^2_{\text{fonte de alimentação}}}{2 \times MTTR_{\text{fonte de alimentação}}}$$

Usando os números de MTTF anteriores, se considerarmos que são necessárias em média 24 horas para um operador humano notar que uma fonte de alimentação falhou e substituí-la, a confiabilidade do par de fontes de alimentação tolerante a falhas é

$$MTTF_{\text{par de fontes de alimentação}} = \frac{MTTF^2_{\text{fonte de alimentação}}}{2 \times MTTR_{\text{fonte de alimentação}}} = \frac{200.000^2}{2 \times 24} \cong 830.000.000$$

tornando o par cerca de 4.150 vezes mais confiável do que uma única fonte de alimentação.

Tendo quantificado o custo, a alimentação e a dependência da tecnologia de computadores, estamos prontos para quantificar o desempenho.

1.8 MEDIÇÃO, RELATÓRIO E RESUMO DO DESEMPENHO

Quando dizemos que um computador é mais rápido do que outro, o que isso significa? O usuário de um computador desktop pode afirmar que um computador é mais rápido quando um programa roda em menos tempo, enquanto um administrador do Amazon.com pode dizer que um computador é mais rápido quando completa mais transações por hora. O usuário do computador está interessado em reduzir o *tempo de resposta* — o tempo entre o início e o término de um evento —, também conhecido como *tempo de execução*. O administrador de um grande centro de processamento de dados pode estar interessado em aumentar o *throughput* — a quantidade total de trabalho feito em determinado tempo.

Comparando as alternativas de projeto, normalmente queremos relacionar o desempenho de dois computadores diferentes, digamos, X e Y. A frase "X é mais rápido do que Y" é usada aqui para significar que o tempo de resposta ou o tempo de execução é inferior em X em relação a Y para determinada tarefa. Em particular, "X é *n* vezes mais rápido do que Y" significará:

$$\frac{\text{Tempo de execução}_Y}{\text{Tempo de execução}_X} = n$$

Como o tempo de execução é o recíproco do desempenho, existe o seguinte relacionamento:

$$n = \frac{\text{Tempo de execução}_Y}{\text{Tempo de execução}_X} = \frac{\dfrac{1}{\text{Desempenho}_Y}}{\dfrac{1}{\text{Desempenho}_X}} = \frac{\text{Desempenho}_X}{\text{Desempenho}_Y}$$

A frase "O throughput de X é 1,3 vez maior que Y" significa que o número de tarefas completadas por unidade de tempo no computador X é 1,3 vez o número completado em Y.

Infelizmente, o tempo nem sempre é a métrica cotada em comparação com o desempenho dos computadores. Nossa posição é de que a única medida consistente e confiável do desempenho é o tempo de execução dos programas reais, e todas as alternativas propostas para o tempo como medida ou aos programas reais como itens medidos por fim levaram a afirmações enganosas ou até mesmo a erros no projeto do computador.

Até mesmo o tempo de execução pode ser definido de diferentes maneiras, dependendo do que nós contamos. A definição mais direta do tempo é chamada *tempo de relógio de parede*, *tempo de resposta* ou *tempo decorrido*, que é a latência para concluir uma tarefa, incluindo acessos ao disco e à memória, atividades de entrada/saída, overhead do sistema operacional — tudo. Com a multiprogramação, o processador trabalha em outro programa enquanto espera pela E/S e pode não minimizar necessariamente o tempo decorrido de um programa. Logo, precisamos de um termo para considerar essa atividade. O *tempo de CPU* reconhece essa distinção e significa o tempo que o processador está computando, *não* incluindo o tempo esperando por E/S ou executando outros programas (é claro que o tempo de resposta visto pelo usuário é o tempo decorrido do programa, e não o tempo de CPU).

Os usuários que executam rotineiramente os mesmos programas seriam os candidatos perfeitos para avaliar um novo computador. Para fazer isso, os usuários simplesmente comparariam o tempo de execução de suas *cargas de trabalho* — a mistura de programas e comandos do sistema operacional que os usuários executam em um computador. Porém, poucos estão nessa situação feliz. A maioria precisa contar com outros métodos para avaliar computadores — e normalmente outros avaliadores —, esperando que esses métodos prevejam o desempenho para o uso do novo computador.

Benchmarks

A melhor escolha de benchmarks para medir o desempenho refere-se a aplicações reais, como o Google Goggles da Seção 1.1. As tentativas de executar programas muito mais simples do que uma aplicação real levaram a armadilhas de desempenho. Alguns exemplos são:

- *kernels*, que são pequenas partes-chave das aplicações reais
- *programas de brinquedo*, que são programas de 100 linhas das primeiras tarefas de programação, como o quicksort
- *benchmarks sintéticos*, que são programas inventados para tentar combinar o perfil e o comportamento de aplicações reais, como o Dhrystone

Hoje, os três estão desacreditados, porque o projetista/arquiteto do compilador pode conspirar para fazer com que o computador pareça mais rápido nesses programas do que em aplicações reais. Infelizmente para os autores deste livro, que, na quarta edição, derrubaram a falácia sobre usar programas sintéticos para caracterizar o desempenho achando que os arquitetos de programas concordavam que ela era indiscutível, o programa sintético Dhrystone ainda é o benchmark mais mencionado para processadores embarcados!

Outra questão está relacionada com as condições nas quais os benchmarks são executados. Um modo de melhorar o desempenho de um benchmark tem sido usar flags específicos de benchmark. Esses flags normalmente causavam transformações que seriam ilegais em muitos programas ou prejudicariam o desempenho de outros. Para restringir esse processo e aumentar a significância dos resultados, os desenvolvedores de benchmark normalmente exigem que o vendedor use um compilador e um conjunto de flags para todos os programas na mesma linguagem (C ou C++). Além dos flags (opções) do compilador, outra questão é a que se refere à permissão das modificações do código-fonte. Existem três técnicas diferentes para resolver essa questão:

1. Nenhuma modificação do código-fonte é permitida.
2. As modificações do código-fonte são permitidas, mas basicamente impossíveis. Por exemplo, os benchmarks de banco de dados contam com programas de banco de dados padrão, que possuem dezenas de milhões de linhas de código. As empresas de banco de dados provavelmente não farão mudanças para melhorar o desempenho de determinado computador.
3. Modificações de fonte são permitidas, desde que a versão modificada produza a mesma saída.

O principal problema que os projetistas de benchmark enfrentam ao permitir a modificação do fonte é se ela refletirá a prática real e oferecerá ideias úteis aos usuários ou se simplesmente reduzirá a precisão dos benchmarks como previsões do desempenho real.

Para contornar o risco de "colocar muitos ovos em uma cesta", séries de aplicações de benchmark, chamadas *pacotes de benchmark*, são uma medida popular do desempenho dos processadores com uma variedade de aplicações. Naturalmente, esses pacotes são tão bons quanto os benchmarks individuais constituintes. Apesar disso, uma vantagem importante desses pacotes é que o ponto fraco de qualquer benchmark é reduzido pela presença de outros benchmarks. O objetivo de um pacote desse tipo é caracterizar o desempenho relativo dos dois computadores, particularmente para programas não incluídos, que os clientes provavelmente usarão.

Para dar um exemplo cauteloso, o EDN Embedded Microprocessor Benchmark Consortium (ou EEMBC) é um conjunto de 41 kernels usados para prever o desempenho de diferentes aplicações embarcadas: automotiva/industrial, consumidor, redes, automação de escritórios e telecomunicações. O EEMBC informa o desempenho não modificado e o desempenho "fúria total", em que quase tudo entra. Por utilizar kernels, e devido às operações de relacionamento, o EEMBC não tem a reputação de ser uma boa previsão de desempenho relativo de diferentes computadores embarcados em campo. O programa sintético Dhrystone, que o EEMBC estava tentando substituir, ainda é relatado em alguns círculos embarcados.

Uma das tentativas mais bem-sucedidas para criar pacotes de aplicação de benchmark padronizadas foi a SPEC (Standard Performance Evaluation Corporation), que teve suas raízes nos esforços do final da década de 1980 para oferecer melhores benchmarks para estações de trabalho. Assim como o setor de computador tem evoluído com o tempo, também evoluiu a necessidade de diferentes pacotes de benchmark — hoje existem benchmarks SPEC para abranger diferentes classes de aplicação. Todos os pacotes de benchmark SPEC e seus resultados relatados são encontrados em <www.spec.org>.

Embora o enfoque de nossa análise seja nos benchmarks SPEC em várias das seções seguintes, também existem muitos benchmarks desenvolvidos para PCs rodando o sistema operacional Windows.

Benchmarks de desktop

Os benchmarks de desktop são divididos em duas classes amplas: benchmarks com uso intensivo do processador e benchmarks com uso intensivo de gráficos, embora muitos

benchmarks gráficos incluam atividade intensa do processador. Originalmente, a SPEC criou um conjunto de benchmarks enfocando o desempenho do processador (inicialmente chamado SPEC89), que evoluiu para sua quinta geração: SPEC CPU2006, que vem após SPEC2000, SPEC95, SPEC92 e SPEC89. O SPEC CPU2006 consiste em um conjunto de 12 benchmarks inteiros (CINT2006) e 17 benchmarks de ponto flutuante (CFP2006). A Figura 1.16 descreve os benchmarks SPEC atuais e seus ancestrais.

	Nome de benchmark em geração SPEC				
Descrição no benchmark	**SPEC2006**	**SPEC2000**	**SPEC95**	**SPEC92**	**SPEC89**
Compilador C GNU	←	←	←	←	gcc
Processamento de string interpretado	←	←	perl		espresso
Otimização combinatória	←	mcf			li
Compactação com classificação em bloco	←	bzip2		compress	eqntott
Jogo Go (IA)	go	vortex	go	sc	
Compactação de vídeo	h264avc	gzip	ijpeg		
Jogos/descoberta de caminho	astar	eon	m88ksim		
Pesquisa de sequência de gene	hmmer	twolf			
Simulação de computador quântico	libquantum	vortex			
Biblioteca de simulação de evento discreto	omnetpp	vpr			
Jogo de xadrez (IA)	sjeng	crafty			
Análise de XML	xalancbmk	parser			
Ondas de CFD/explosão	bwaves				fpppp
Relatividade numérica	cactusADM				tomcatv
Código de elemento finito	calculix				doduc
Estrutura de solucionador de equação diferencial	dealII				nasa7
Química quântica	gamess				spice
Solucionador EM (domínio de frequência/tempo)	GemsFDTD			swim	matrix300
Dinâmica molecular escalável (~NAMD)	gromacs		apsi	hydro2d	
Método Lattice Boltzman (fluxo de fluido/ar)	lbm		mgrid	su2cor	
Simulação de grande turbilhão/CFD turbulento	LESlie3d	wupwise	applu	wave5	
Cromodinâmica quântica	milc	apply	turb3d		
Dinâmica molecular	namd	galgel			
Rastreamento de raio de imagem	povray	mesa			
Álgebra linear dispersa	soplex	art			
Reconhecimento de voz	sphinx3	equake			
Química quântica/orientação a objeto	tonto	facerec			
Pesquisa e previsão de tempo	wrf	ammp			
Hidrodinâmica magnética (astrofísica)	zeusmp	lucas			
		fma3d			
		sixtrack			

FIGURA 1.16 Programas do SPEC2006 e a evolução dos benchmarks SPEC com o tempo, com programas inteiros na parte superior e programas de ponto flutuante na parte inferior.

Dos 12 programas inteiros do SPEC2006, nove são escritos em C e o restante em C++. Para os programas de ponto flutuante, a composição é de seis em FORTRAN, quatro em C++, três em C e quatro misturados entre C e Fortran. A figura mostra os 70 programas nas versões de 1989, 1992, 1995, 2000 e 2006. As descrições de benchmark, à esquerda, são apenas para o SPEC2006 e não se aplicam às anteriores. Os programas na mesma linha de diferentes gerações do SPEC geralmente não estão relacionados; por exemplo, fpppp não é o código CFD como bwaves. Gcc é o mais antigo do grupo. Somente três programas inteiros e três programas de ponto flutuante são novos para o SPEC2006. Embora alguns sejam levados de uma geração para outra, a versão do programa muda e a entrada ou o tamanho do benchmark normalmente é alterado para aumentar seu tempo de execução e evitar perturbação na medição ou domínio do tempo de execução por algum fator diferente do tempo de CPU.

Os benchmarks SPEC são programas reais modificados para serem portáveis e minimizar o efeito da E/S sobre o desempenho. Os benchmarks inteiros variam de parte de um compilador C até um programa de xadrez ou uma simulação de computador quântico. Os benchmarks de ponto flutuante incluem códigos de grade estruturados para modelagem de elemento finito, códigos de método de partícula para dinâmica molecular e códigos de álgebra linear dispersa para dinâmica de fluidos. O pacote SPEC CPU é útil para o benchmarking de processador para sistemas de desktop e servidores de único processador. Veremos os dados sobre muitos desses programas no decorrer deste capítulo. Entretanto, observe que esses programas compartilham pouco com linguagens de programação e ambientes e o Google Goggles, que a Seção 1.1 descreve. O sete usa C++, o oito usa C e o nove usa Fortran! Eles estão até ligados estatisticamente, e os próprios aplicativos são simples. Não está claro que o SPENCINT2006 e o SPECFP2006 capturam o que é excitante sobre a computação no século XXI.

Na Seção 1.11, descrevemos as armadilhas que têm ocorrido no desenvolvimento do pacote de benchmark SPEC, além dos desafios na manutenção de um pacote de benchmark útil e previsível.

O SPEC CPU2006 visa ao desempenho do processador, mas o SPEC oferece muitos outros benchmarks.

Benchmarks de servidor

Assim como os servidores possuem funções múltiplas, também existem múltiplos tipos de benchmark. O benchmark mais simples talvez seja aquele orientado a throughput do processador. O SPEC CPU2000 usa os benchmarks SPEC CPU para construir um benchmark de throughput simples, em que a taxa de processamento de um multiprocessador pode ser medida pela execução de várias cópias (normalmente tantas quanto os processadores) de cada benchmark SPEC CPU e pela conversão do tempo de CPU em uma taxa. Isso leva a uma medida chamada SPECrate, que é uma medida de paralelismo em nível de requisição, da Seção 1.2. Para medir o paralelismo de nível de thread, o SPEC oferece o que chamamos benchmarks de computação de alto desempenho com o OpenMP e o MPI.

Além da SPECrate, a maioria das aplicações de servidor e benchmarks possui atividade significativa de E/S vinda do disco ou do tráfego de rede, incluindo benchmarks para sistemas de servidor de arquivos, para servidores Web e para sistemas de banco de dados e processamento de transação. O SPEC oferece um benchmark de servidor de arquivos (SPECSFS) e um benchmark de servidor Web (SPECWeb). O SPECSFS é um benchmark para medir o desempenho do NFS (Network File System) usando um script de solicitações ao servidor de arquivos; ele testa o desempenho do sistema de E/S (tanto E/S de disco quanto de rede), além do processador. O SPECSFS é um benchmark orientado a throughput, mas com requisitos importantes de tempo de resposta (o Apêndice D tratará detalhadamente de alguns benchmarks de arquivo e do sistema de E/S). O SPECWeb é um benchmark de servidor Web que simula vários clientes solicitando páginas estáticas e dinâmicas de um servidor, além dos clientes postando dados no servidor. O SPECjbb mede o desempenho do servidor para aplicativos Web escritos em Java. O benchmark SPEC mais recente é o SPECvirt_Sc2010, que avalia o desempenho end-to-end de servidores virtualizados de data center incluindo hardware, a camada de máquina virtual e o sistema operacional virtualizado. Outro benchmark SPEC recente mede a potência, que examinaremos na Seção 1.10.

Os benchmarks de processamento de transação (Transaction-Processing — TP) medem a capacidade de um sistema para lidar com transações, que consistem em acessos e atualizações de banco de dados. Os sistemas de reserva aérea e os sistemas de terminal bancário são exemplos simples típicos de TP; sistemas de TP mais sofisticados envolvem bancos de dados complexos e tomada de decisão. Em meados da década de 1980, um grupo de

engenheiros interessados formou o Transaction Processing Council (TPC) independente de fornecedor, para tentar criar benchmarks realistas e imparciais para TP. Os benchmarks do TPC são descritos em <www.tpc.org>.

O primeiro benchmark TPC, TPC-A, foi publicado em 1985 e desde então tem sido substituído e aprimorado por vários benchmarks diferentes. O TPC-C, criado inicialmente em 1992, simula um ambiente de consulta complexo. O TPC-H molda o suporte à decisão ocasional — as consultas não são relacionadas e o conhecimento de consultas passadas não pode ser usado para otimizar consultas futuras. O TCP-E é uma nova carga de trabalho de processamento de transação on-line (On-Line Transaction Processing — OLTP) que simula as contas dos clientes de uma firma de corretagem. O esforço mais recente é o TPC Energy, que adiciona métricas de energia a todos os benchmarks TPC existentes.

Todos os benchmarks TPC medem o desempenho em transações por segundo. Além disso, incluem um requisito de tempo de resposta, de modo que o desempenho do throughput é medido apenas quando o limite de tempo de resposta é atendido. Para modelar sistemas do mundo real, taxas de transação mais altas também estão associadas a sistemas maiores, em termos de usuários e do banco de dados ao qual as transações são aplicadas. Finalmente, cabe incluir o custo do sistema para um sistema de benchmark, permitindo comparações precisas de custo-desempenho. O TPC modificou sua política de preços para que exista uma única especificação para todos os benchmarks TPC e para permitir a verificação dos preços que a TPC publica.

Reportando resultados de desempenho

O princípio orientador dos relatórios das medições de desempenho deve ser a propriedade de serem *reproduzíveis* — listar tudo aquilo de que outro experimentador precisaria para duplicar os resultados. Um relatório de benchmark SPEC exige uma descrição extensa do computador e dos flags do compilador, além da publicação da linha de referência e dos resultados otimizados. Além das descrições de parâmetros de ajuste de hardware, software e linha de referência, um relatório SPEC contém os tempos de desempenho reais, mostrados tanto em formato de tabulação quanto como gráfico. Um relatório de benchmark TPC é ainda mais completo, pois precisa incluir resultados de uma auditoria de benchmarking e informação de custo. Esses relatórios são excelentes fontes para encontrar o custo real dos sistemas de computação, pois os fabricantes competem em alto desempenho e no fator custo-desempenho.

Resumindo resultados do desempenho

No projeto prático do computador, você precisa avaliar milhares de opções de projeto por seus benefícios quantitativos em um pacote de benchmarks que acredita ser relevante. Da mesma forma, os consumidores que tentam escolher um computador contarão com medidas de desempenho dos benchmarks, que esperam ser semelhantes às aplicações do usuário. Nos dois casos é útil ter medições para um pacote de benchmarks de modo que o desempenho das aplicações importantes seja semelhante ao de um ou mais benchmarks desse pacote e que a variabilidade no desempenho possa ser compreendida. No caso ideal, o pacote se parece com uma amostra estatisticamente válida do espaço da aplicação, mas requer mais benchmarks do que normalmente são encontrados na maioria dos pacotes, exigindo uma amostragem aleatória que quase nenhum pacote de benchmark utiliza.

Depois que escolhermos medir o desempenho com um pacote de benchmark, gostaríamos de poder resumir os resultados desse desempenho em um único número. Uma técnica simples para o cálculo de um resultado resumido seria comparar as médias aritméticas dos tempos de execução dos programas no pacote. Infelizmente, alguns programas SPEC gastam quatro vezes mais tempo do que outros, de modo que esses programas seriam muito mais

importantes se a média aritmética fosse o único número utilizado para resumir o desempenho. Uma alternativa seria acrescentar um fator de peso a cada benchmark e usar a média aritmética ponderada como único número para resumir o desempenho. O problema seria, então, como selecionar os pesos. Como a SPEC é um consórcio de empresas concorrentes, cada empresa poderia ter seu próprio conjunto favorito de pesos, o que tornaria difícil chegar a um consenso. Uma solução é usar pesos que façam com que todos os programas executem por um mesmo tempo em algum computador de referência, mas isso favorece os resultados para as características de desempenho do computador de referência.

Em vez de selecionar pesos, poderíamos normalizar os tempos de execução para um computador de referência, dividindo o tempo no computador de referência pelo tempo no computador que está sendo avaliado, gerando uma razão proporcional ao desempenho. O SPEC utiliza essa técnica, chamando a razão de SPECRatio. Ele possui uma propriedade particularmente útil, que combina o modo como comparamos o desempenho do computador no decorrer deste capítulo, ou seja, comparando razão de desempenho. Por exemplo, suponha que o SPECRatio do computador A em um benchmark tenha sido 1,25 vez maior que o do computador B; então, você saberia que:

$$1,25 = \frac{\text{SPECRatio}_A}{\text{SPECRatio}_B} = \frac{\dfrac{\text{Tempo de execução}_{\text{referência}}}{\text{Tempo de execução}_A}}{\dfrac{\text{Tempo de execução}_{\text{referência}}}{\text{Tempo de execução}_B}} = \frac{\text{Tempo de execução}_B}{\text{Tempo de execução}_A} = \frac{\text{Desempenho}_A}{\text{Desempenho}_B}$$

Observe que os tempos de execução no computador de referência caem e a escolha do computador de referência é irrelevante quando as comparações são feitas como uma razão, que é a técnica que utilizamos coerentemente. A Figura 1.17 apresenta um exemplo.

Benchmarks	Tempo no Ultra 5 (s)	Tempo no Opteron (s)	SPECRatio	Tempo no Itanium 2 (s)	SPECRatio	Tempos no Opteron/Itanium (s)	SPECRatios no Itanium/Opteron
wupwise	1.600	51,5	31,06	56,1	28,53	0,92	0,92
swim	3.100	125	24,73	70,7	43,85	1,77	1,77
mgrid	1.800	98	18,37	65,8	27,36	1,49	1,49
applu	2.100	94	22,34	50,9	41,25	1,85	1,85
mesa	1.400	64,6	21,69	108	12,99	0,6	0,6
galgel	2.900	86,4	33,57	40	72,47	2,16	2,16
art	2.600	92,4	28,13	21	123,67	4,4	4,4
equake	1.300	72,6	17,92	36,3	35,78	2	2
facerec	1.900	73,6	25,8	86,9	21,86	0,85	0,85
ammp	2.200	136	16,14	132	16,63	1,03	1,03
lucas	2.000	88,8	22,52	107	18,76	0,83	0,83
fma3d	2.100	120	17,48	131	16,09	0,92	0,92
sixtrack	1.100	123	8,95	68,8	15,99	1,79	1,79
apsi	2.600	150	17,36	231	11,27	0,65	0,65
Média geométrica			20,86		27,12	1,3	1,3

FIGURA 1.17 Tempos de execução do SPECfp2000 (em segundos) para o Sun Ultra 5 — o computador de referência do SPEC2000 — e tempos de execução e SPECRatios para o AMD Opteron e Intel Itanium 2.

(O SPEC2000 multiplica a razão dos tempos de execução por 100 para remover as casas decimais do resultado, de modo que 20,86 é informado como 2.086.) As duas últimas colunas mostram as razões dos tempos de execução e SPECRatios. Esta figura demonstra a irrelevância do computador de referência no desempenho relativo. A razão dos tempos de execução é idêntica à razão dos SPECRatios, e a razão da média geométrica (27,12/20,86 = 1,30) é idêntica à média geométrica das razões (1,3).

Como um SPECRatio é uma razão, e não um tempo de execução absoluto, a média precisa ser calculada usando a *média geométrica* (como os SPECRatios não possuem unidades, a comparação de SPECRatios aritmeticamente não tem sentido). A fórmula é:

$$\text{Medida geométrica} = \sqrt[n]{\prod_{i=1}^{n} \text{amostra}_1}$$

No caso do SPEC, $amostra_i$ é o SPECRatio para o programa *i*. O uso da média geométrica garante duas propriedades importantes:

1. A média geométrica das razões é igual à razão das médias geométricas.
2. A razão das médias geométricas é igual à média geométrica das razões de desempenho, o que implica que a escolha do computador de referência é irrelevante.

Logo, as motivações para usar a média geométrica são substanciais, especialmente quando usamos razões de desempenho para fazer comparações.

Exemplo Mostre que a razão das médias geométricas é igual à média geométrica das razões de desempenho e que a comunicação de referência do SPECRatio não importa.

Resposta Considere dois computadores, A e B, e um conjunto de SPECRatios para cada um.

$$\frac{\text{Medida geométrica}_A}{\text{Medida geométrica}_B} = \frac{\sqrt[n]{\prod_{i=1}^{n} \text{SPECRatio A}_i}}{\sqrt[n]{\prod_{i=1}^{n} \text{SPECRatio B}_i}} = \sqrt[n]{\prod_{i=1}^{n} \frac{\text{SPECRatio A}_i}{\text{SPECRatio B}_i}}$$

$$= \sqrt[n]{\prod_{i=1}^{n} \frac{\frac{\text{Tempo de execução}_{\text{referência}_i}}{\text{Tempo de execução}_{A_i}}}{\frac{\text{Tempo de execução}_{\text{referência}_i}}{\text{Tempo de execução}_{B_i}}}} = \sqrt[n]{\prod_{i=1}^{n} \frac{\text{Tempo de execução}_{B_i}}{\text{Tempo de execução}_{A_i}}} = \sqrt[n]{\prod_{i=1}^{n} \frac{\text{Desempenho}_{A_i}}{\text{Desempenho}_{B_i}}}$$

Ou seja, a razão das médias geométricas dos SPECRatios de A e B é a média geométrica das razões de desempenho de A para B de todos os benchmarks no pacote. A Figura 1.17 demonstra a validade usando exemplos da SPEC.

1.9 PRINCÍPIOS QUANTITATIVOS DO PROJETO DE COMPUTADORES

Agora que vimos como definir, medir e resumir desempenho, custo, dependência e potência, podemos explorar orientações e princípios que são úteis no projeto e na análise de computadores. Esta seção introduz observações importantes sobre projeto, além de duas equações para avaliar alternativas.

Tire proveito do paralelismo

Tirar proveito do paralelismo é um dos métodos mais importantes para melhorar o desempenho. Cada capítulo deste livro apresenta um exemplo de como o desempenho é melhorado por meio da exploração do paralelismo. Oferecemos três exemplos rápidos, que serão tratados mais amplamente em outros capítulos.

Nosso primeiro exemplo é o uso do paralelismo em nível do sistema. Para melhorar o desempenho de throughput em um benchmark de servidor típico, como SPECWeb ou TPC-C, vários processadores e múltiplos discos podem ser usados. A carga de trabalho

de tratar com solicitações pode, portanto, ser distribuída entre os processadores e discos, resultando em um throughput melhorado. Ser capaz de expandir a memória e o número de processadores e discos é o que chamamos *escalabilidade*, e constitui um bem valioso para os servidores. A disrtribuição dos dados por vários discos para leituras e gravações paralelas habilita o paralelismo em nível de dados. O SPECWeb também depende do paralelismo em nível de requisição para usar muitos processadores, enquanto o TPC-C usa paralelismo em nível de thread para o processamento mais rápido de pesquisas em bases de dados.

Em nível de um processador individual, tirar proveito do paralelismo entre as instruções é crucial para conseguir alto desempenho. Uma das maneiras mais simples de fazer isso é por meio do pipelining (isso será explicado com detalhes no Apêndice C, e é o foco principal do Capítulo 3). A ideia básica por trás do pipelining é sobrepor a execução da instrução para reduzir o tempo total a fim de completar uma sequência de instruções. Um *insight* importante que permite que o pipelining funcione é que nem toda instrução depende do seu predecessor imediato; portanto, executar as instruções completa ou parcialmente em paralelo é possível. O pipelining é o exemplo mais conhecido de paralelismo em nível de instrução.

O paralelismo também pode ser explorado no nível de projeto digital detalhado. Por exemplo, as caches associadas por conjunto utilizam vários bancos de memória que normalmente são pesquisados em paralelo para se encontrar um item desejado. As unidades lógica e aritmética (Arithmetic-Logical Units — ALUs) modernas utilizam carry-lookahead, que usa o paralelismo para acelerar o processo de cálculo de somas de linear para logarítmico no número de bits por operando. Esses são mais exemplos de paralelismo em nível de dados.

Princípio de localidade

Observações fundamentais importantes vêm das propriedades dos programas. A propriedade dos programas mais importante que exploramos regularmente é o *princípio de localidade*: os programas costumam reutilizar dados e instruções que usaram recentemente. Uma regra prática bastante aceita é de que um programa gasta 90% de seu tempo de execução em apenas 10% do código. Uma aplicação desse princípio é a possibilidade de prever com razoável precisão as instruções e os dados que um programa usará num futuro próximo com base em seus acessos num passado recente. O princípio de localidade também se aplica aos acessos a dados, embora não tão fortemente quanto aos acessos ao código.

Dois tipos diferentes de localidade têm sido observados. No tocante à *localidade temporal*, é provável que os itens acessados recentemente sejam acessados num futuro próximo. A *localidade espacial* afirma que os itens cujos endereços estão próximos um do outro costumam ser referenciados em curto espaço de tempo. Veremos esses princípios aplicados no Capítulo 2.

Foco no caso comum

Talvez o princípio mais importante e penetrante do projeto de computador seja focar no caso comum: ao fazer uma escolha de projeto, favoreça o caso frequente em vez do caso pouco frequente. Esse princípio se aplica à determinação de como gastar recursos, pois o impacto da melhoria será mais alto se a ocorrência for frequente.

Focar no caso comum funciona tanto para a potência como para os recursos de alocação e desempenho. As unidades de busca e decodificação de instruções de um processador pode ser usada com muito mais frequência do que um multiplicador, por isso deve ser otimizada primeiro. Isso também funciona na dependência. Se um servidor de banco de dados possui 50 discos para cada processador, como na próxima seção, a dependência de armazenamento dominará a dependência do sistema.

Além disso, o caso frequente normalmente é mais simples e pode ser feito com mais rapidez do que o caso pouco frequente. Por exemplo, ao somar dois números no processador, podemos esperar que o estouro (overflow) seja uma circunstância rara e, assim, podemos melhorar o desempenho, otimizando o caso mais comum, ou seja, sem nenhum estouro. Isso pode atrasar o caso em que ocorre estouro, mas, se isso for raro, o desempenho geral será melhorado, otimizando o processador para o caso normal.

Veremos muitos casos desse princípio em todo este capítulo. Na aplicação desse princípio simples, temos que decidir qual é o caso frequente e quanto desempenho pode ser melhorado tornando-o mais rápido. Uma lei fundamental, chamada *lei de Amdahl*, pode ser usada para quantificar esse princípio.

Lei de Amdahl

O ganho de desempenho obtido com a melhoria de alguma parte de um computador pode ser calculado usando a lei de Amdahl. Essa lei estabelece que a melhoria de desempenho a ser conseguida com o uso de algum modo de execução mais rápido é limitada pela fração de tempo que o modo mais rápido pode ser usado.

A lei de Amdahl define o *ganho de velocidade*, que pode ser obtido usando-se um recurso em particular. O que é ganho de velocidade? Suponha que possamos fazer uma melhoria em um computador que aumentará seu desempenho quando ele for usado. O ganho de velocidade *é a razão*:

$$\text{Ganho de velocidade} = \frac{\text{Desempenho para a tarefa inteira usando a melhoria quando possível}}{\text{Desempenho para a tarefa inteira sem usar a melhoria}}$$

Como alternativa,

$$\text{Ganho de velocidade} = \frac{\text{Desempenho para a tarefa inteira sem usar a melhoria}}{\text{Desempenho para a tarefa inteira usando a melhoria quando possível}}$$

O ganho de velocidade nos diz quão mais rápido uma tarefa rodará usando o computador com a melhoria em vez do computador original.

A lei de Amdahl nos dá um modo rápido de obter o ganho de velocidade a partir de alguma melhoria, o que depende de dois fatores:

1. *A fração do tempo de computação no computador original que pode ser convertida para tirar proveito da melhoria.* Por exemplo, se 20 segundos do tempo de execução de um programa que leva 60 segundos no total puderem usar uma melhoria, a fração será 20/60. Esse valor, que chamaremos de $\text{Fração}_{\text{melhorada}}$, será sempre menor ou igual a 1.
2. *A melhoria obtida pelo modo de execução melhorado, ou seja, quão mais rápido a tarefa seria executada se o modo melhorado fosse usado para o programa inteiro.* Esse valor é o tempo do modo original sobre o tempo do modo melhorado. Se o modo melhorado levar, digamos, 2 segundos para uma parte do programa, enquanto é de 5 segundos no modo original, a melhoria será de 5/2. Chamaremos esse valor, que é sempre maior que 1, de $\text{Ganho de velocidade}_{\text{melhorado}}$.

O tempo de execução usando o computador original com o modo melhorado será o tempo gasto usando a parte não melhorada do computador mais o tempo gasto usando a melhoria:

$$\text{Tempo de execução}_{\text{novo}} = \text{Tempo de execução}_{\text{antigo}} \times \left[(1 - \text{Fração}_{\text{melhorada}}) + \frac{\text{Fração}_{\text{melhorada}}}{\text{Ganho de velocidade}_{\text{melhorado}}} \right]$$

O ganho de velocidade geral é a razão dos tempos de execução:

$$\text{Ganho de velocidade}_{\text{geral}} = \frac{\text{Tempo de execução}_{\text{antigo}}}{\text{Tempo de execução}_{\text{novo}}} = \frac{1}{(1 - \text{Fração}_{\text{melhorada}}) + \dfrac{\text{Fração}_{\text{melhorada}}}{\text{Ganho de velocidade}_{\text{melhorado}}}}$$

Exemplo Suponha que queiramos melhorar o processador usado para serviço na Web. O novo processador é 10 vezes mais rápido em computação na aplicação de serviço da Web do que o processador original. Considerando que o processador original está ocupado com cálculos 40% do tempo e esperando por E/S 60% do tempo, qual é o ganho de velocidade geral obtido pela incorporação da melhoria?

Resposta $\text{Fração}_{\text{melhorada}} = 0{,}4; \text{Ganho de velocidade}_{\text{melhorado}} = 10;$

$$\text{Ganho de velocidade}_{\text{geral}} = \frac{1}{0{,}6 + \dfrac{0{,}4}{10}} = \frac{1}{0{,}64} \approx 1{,}56$$

A lei de Amdahl expressa os retornos diminuídos: a melhoria incremental no ganho de velocidade obtida pela melhoria de apenas uma parte da computação diminui à medida que as melhorias são acrescentadas. Um corolário importante da lei de Amdahl é que, se uma melhoria só for utilizável por uma fração de uma tarefa, não poderemos agilizar essa tarefa mais do que o inverso de 1 menos essa fração.

Um engano comum na aplicação da lei de Amdahl é confundir "fração de tempo convertida *para usar uma melhoria*" com "fração de tempo *após a melhoria estar em uso*". Se, em vez de medir o tempo que *poderíamos usar* a melhoria em um cálculo, medirmos o tempo *após* a melhoria estar em uso, os resultados serão incorretos!

A lei de Amdahl pode servir de guia para o modo como uma melhoria incrementará o desempenho e como distribuir recursos para melhorar o custo-desempenho. O objetivo, claramente, é investir recursos proporcionais onde o tempo é gasto. A lei de Amdahl é particularmente útil para comparar o desempenho geral do sistema de duas alternativas, mas ela também pode ser aplicada para comparar duas alternativas de um projeto de processador, como mostra o exemplo a seguir.

Exemplo Uma transformação comum exigida nos processadores de gráficos é a raiz quadrada. As implementações de raiz quadrada com ponto flutuante (PF) variam muito em desempenho, sobretudo entre processadores projetados para gráficos. Suponha que a raiz quadrada em PF (FPSQR) seja responsável por 20% do tempo de execução de um benchmark gráfico crítico. Uma proposta é melhorar o hardware de FPSQR e agilizar essa operação por um fator de 10. A outra alternativa é simplesmente tentar fazer com que todas as operações de PF no processador gráfico sejam executadas mais rapidamente por um fator de 1,6; as instruções de PF são responsáveis por metade do tempo de execução para a aplicação. A equipe de projeto acredita que pode fazer com que todas as instruções de PF sejam executadas 1,6 vez mais rápido com o mesmo esforço exigido para a raiz quadrada rápida. Compare essas duas alternativas de projeto.

Resposta Podemos comparar essas alternativas comparando os ganhos de velocidade:

$$\text{Ganho de velocidade}_{\text{FPSQR}} = \frac{1}{(1 - 0{,}2) + \dfrac{0{,}2}{10}} = \frac{1}{0{,}82} = 1{,}22$$

$$\text{Ganho de velocidade}_{\text{FP}} = \frac{1}{(1 - 0{,}5) + \dfrac{0{,}5}{1{,}6}} = \frac{1}{0{,}8125} = 1{,}23$$

Melhorar o ganho de velocidade das operações de PF em geral é ligeiramente melhor devido à frequência mais alta.

A lei de Amdahl se aplica além do desempenho. Vamos refazer o exemplo de confiabilidade da páginas 31 e 32 depois de melhorar a confiabilidade da fonte de alimentação, por meio da redundância, de 200.000 horas para 830.000.000 horas MTTF ou 4.150 vezes melhor.

Exemplo O cálculo das taxas de falha do subsistema de disco foi

$$\begin{aligned}\text{Taxa de falha}_{\text{sistema}} &= 10 \times \frac{1}{1.000.000} + \frac{1}{500.000} + \frac{1}{200.000} + \frac{1}{200.000} + \frac{1}{1.000.000} \\ &= \frac{10+2+5+5+1}{1.000.000 \text{ horas}} = \frac{23}{1.000.000 \text{ horas}}\end{aligned}$$

Portanto, a fração da taxa de falha que poderia ser melhorada é 5 por milhão de horas, das 23 para o sistema inteiro, ou 0,22.

Resposta A melhoria de confiabilidade seria

$$\text{Melhoria}_{\text{par de fontes}} = \frac{1}{(1-0,22) + \frac{0,22}{4150}} = \frac{1}{0,78} = 1,28$$

Apesar de uma impressionante melhoria de 4.150 vezes na confiabilidade de um módulo, do ponto de vista do sistema a mudança possui um benefício mensurável, porém pequeno.

Nos exemplos precedentes, precisamos da fração consumida pela versão nova e melhorada; costuma ser difícil medir esses tempos diretamente. Na seção seguinte, veremos outra forma de fazer essas comparações com base no uso de uma equação que decompõe o tempo de execução da CPU em três componentes separados. Se soubermos como uma alternativa afeta esses componentes, poderemos determinar seu desempenho geral. Normalmente é possível montar simuladores que medem esses componentes antes que o hardware seja realmente projetado.

A equação de desempenho do processador

Basicamente todos os computadores são construídos usando um clock que trabalha a uma taxa constante. Esses eventos de tempo discretos são chamados de *ticks*, *ticks de clock*, *períodos de clock*, *clocks*, *ciclos* ou *ciclos de clock*. Os projetistas de computador referem-se ao tempo de um período de clock por sua duração (por exemplo, 1 ns) ou por sua frequência (por exemplo, 1 GHz). O tempo de CPU para um programa pode, então, ser expresso de duas maneiras:

$$\text{Tempo de CPU} = \text{Ciclos de clock de CPU para um programa} \times \text{Tempo do ciclo de clock}$$

ou

$$\text{Tempo de CPU} = \frac{\text{Ciclos de clock de CPU para um programa}}{\text{Frequência de clock}}$$

Além do número de ciclos de clock necessários para executar um programa, também podemos contar o número de instruções executadas — o *tamanho do caminho de instrução* ou *número de instruções* (Instruction Count — IC). Se soubermos o número de ciclos de clock e o contador de instruções, poderemos calcular o número médio de *ciclos de clock por instruções* (Clock Cycles Per Instruction — CPI). Por ser mais fácil de trabalhar e porque neste livro lidaremos com processadores simples, usaremos o CPI. Às vezes, os projetistas também usam *instruções por clock* (Instructions Per Clock — IPC), que é o inverso do CPI.

O CPI é calculado como

$$\text{CPI} = \frac{\text{Ciclos de clock de CPU para um programa}}{\text{Número de instruções}}$$

Esse valor de mérito do processador oferece visões para diferentes estilos de conjuntos de instruções e de implementações, e o usaremos bastante nos quatro capítulos seguintes.

Transpondo o número de instruções na fórmula anterior, os ciclos de clock podem ser definidos como IC × CPI. Isso nos permite usar o CPI na fórmula do tempo de execução:

$$\text{Tempo de CPU} = \text{Número de instruções} \times \text{Ciclos por instrução} \times \text{Tempo de ciclo de clock}$$

Expandindo a primeira fórmula para as unidades de medida, vemos como as partes se encaixam:

$$\frac{\text{Instruções}}{\text{Programa}} \times \frac{\text{Ciclos de clock}}{\text{Instrução}} \times \frac{\text{Segundos}}{\text{Ciclos de clock}} = \frac{\text{Segundos}}{\text{Programa}} = \text{Tempo de CPU}$$

Conforme a fórmula demonstra, o desempenho do processador depende de três características: ciclo de clock (ou frequência), ciclos de clock por instruções e número de instruções. Além do mais, o tempo de CPU depende *igualmente* dessas três características: a melhoria de 10% em qualquer um deles leva à melhoria de 10% no tempo de CPU.

Infelizmente, é difícil mudar um parâmetro de modo completamente isolado dos outros, pois as tecnologias básicas envolvidas na mudança de cada característica são interdependentes:

- *Tempo de ciclo de clock* — Tecnologia e organização do hardware
- *CPI* — Organização e arquitetura do conjunto de instruções
- *Número de instruções* — Arquitetura do conjunto de instruções e tecnologia do computador

Por sorte, muitas técnicas potenciais de melhoria de desempenho melhoram principalmente um componente do desempenho do processador, com impactos pequenos ou previsíveis sobre os outros dois.

Às vezes, é útil projetar o processador para calcular o número total de ciclos de clock do processador como

$$\text{Ciclos de clock da CPU} = \sum_{i=1}^{n} \text{IC}_i \times \text{CPI}_i$$

onde IC_i representa o número de vezes que a instrução i é executada em um programa e CPI_i representa o número médio de clocks por instrução para a instrução i. Essa forma pode ser usada para expressar o tempo de CPU como

$$\text{Tempo de CPU} = \left(\sum_{i=1}^{n} \text{IC}_i \times \text{CPI}_i\right) \times \text{Tempo de ciclo de clock}$$

e o CPI geral como

$$\text{CPI} = \frac{\sum_{i=1}^{n} \text{IC}_i \times \text{CPI}_i}{\text{Número de instruções}} = \sum_{i=1}^{n} \frac{\text{IC}_i}{\text{Número de instruções}} \times \text{CPI}_i$$

A última forma do cálculo do CPI utiliza cada CPI_i e a fração de ocorrências dessa instrução em um programa (ou seja, IC_i ÷ número de instruções). O CPI_i deve ser medido, e não apenas calculado a partir de uma tabela no final de um manual de referência, pois precisa levar em consideração os efeitos de pipeline, as faltas de cache e quaisquer outras ineficiências do sistema de memória.

Considere nosso exemplo de desempenho da página 42, modificado aqui para usar medições da frequência das instruções e dos valores de CPI da instrução, que, na prática, são obtidos pela simulação ou pela instrumentação do hardware.

Exemplo Suponha que tenhamos feito as seguintes medições:

Frequência das operações de PF = 25%
CPI médio das operações de PF = 4,0
CPI médio das outras instruções = 1,33
Frequência da FPSQR = 2%
CPI da FPSQR = 20

Considere que as duas alternativas de projeto sejam diminuir o CPI da FPSQR para 2 ou diminuir o CPI médio de todas as operações de PF para 2,5. Compare essas duas alternativas de projeto usando a equação de desempenho do processador.

Resposta Em primeiro lugar, observe que somente o CPI muda; a taxa de clock e o número de instruções permanecem idênticos. Começamos encontrando o CPI original sem qualquer melhoria:

$$\text{CPI}_{original} = \sum_{i=1}^{n} \text{CPI}_i \times \left(\frac{\text{IC}_i}{\text{Número de instruções}} \right)$$
$$= (4 \times 25\%) + (1{,}33 \times 75\%) = 2{,}0$$

Podemos calcular o CPI para a FPSQR melhorada subtraindo os ciclos salvos do CPI original:

$$\text{CPI}_{\text{com nova FPSQR}} = \text{CPI}_{\text{original}} - 2\% \times (\text{CPI}_{\text{FPSQR antiga}} - \text{CPI}_{\text{de nova FPSQR apenas}})$$
$$= 2{,}0 - 2\% \times (20 - 2) = 1{,}64$$

Podemos calcular o CPI para a melhoria de todas as instruções de PF da mesma forma ou somando os CPIs de PF e de não PF. Usando a última técnica, temos

$$\text{CPI}_{\text{novo FP}} = (75\% \times 1{,}33) + (25\% \times 2{,}5) = 1{,}625$$

Como o CPI da melhoria geral de PF é ligeiramente inferior, seu desempenho será um pouco melhor. Especificamente, o ganho de velocidade para a melhoria de PF geral é

$$\text{Ganho de velocidade}_{\text{novo FP}} = \frac{\text{Tempo de CPU}_{\text{original}}}{\text{Tempo de CPU}_{\text{nova FP}}}$$
$$= \frac{\text{IC} \times \text{Ciclo de clock} \times \text{CPi}_{\text{original}}}{\text{IC} \times \text{Ciclo de clock} \times \text{CPi}_{\text{nova FP}}}$$
$$= \frac{\text{CPi}_{\text{original}}}{\text{CPi}_{\text{nova FP}}} = \frac{2{,}00}{1{,}625} = 1{,}23$$

Felizmente, obtemos esse mesmo ganho de velocidade usando a lei de Amdahl na página 41.

Normalmente, é possível medir as partes constituintes da equação de desempenho do processador. Essa é uma vantagem importante do uso dessa equação *versus* a lei de Amdahl no exemplo anterior. Em particular, pode ser difícil medir itens como a fração do tempo de execução pela qual um conjunto de instruções é responsável. Na prática, isso provavelmente seria calculado somando-se o produto do número de instruções e o CPI para cada uma das instruções no conjunto. Como os pontos de partida normalmente são o número de instruções e as medições de CPI, a equação de desempenho do processador é incrivelmente útil.

Para usar a equação de desempenho do processador como uma ferramenta de projeto, precisamos ser capazes de medir os diversos fatores. Para determinado processador, é fácil obter o tempo de execução pela medição, enquanto a velocidade do clock padrão é

conhecida. O desafio está em descobrir o número de instruções ou o CPI. A maioria dos novos processadores inclui contadores para instruções executadas e para ciclos de clock. Com o monitoramento periódico desses contadores, também é possível conectar o tempo de execução e o número de instruções a segmentos do código, o que pode ser útil para programadores que estão tentando entender e ajustar o desempenho de uma aplicação. Em geral, um projetista ou programador desejará entender o desempenho em um nível mais detalhado do que o disponibilizado pelos contadores do hardware. Por exemplo, eles podem querer saber por que o CPI é o que é. Nesses casos, são usadas técnicas de simulação como aquelas empregadas para os processadores que estão sendo projetados.

Técnicas que ajudam na eficiência energética, como escalamento dinâmico de frequência, de voltagem e overclocking (Seção 1.5), tornam essa equação mais difícil de usar, já que a velocidade do clock pode variar enquanto medimos o programa. Uma abordagem simples é desligar esses recursos para tornar os resultados passíveis de reprodução. Felizmente, já que muitas vezes o desempenho e a eficiência energética estão altamente correlacionados — levar menos tempo para rodar um programa geralmente poupa energia —, provavelmente é seguro considerar o desempenho sem se preocupar com o impacto do DVFS ou overclocking sobre os resultados.

1.10 JUNTANDO TUDO: DESEMPENHO E PREÇO-DESEMPENHO

Nas seções "Juntando tudo" que aparecem próximo ao final de cada capítulo, mostramos exemplos reais que utilizam os princípios explicados no capítulo. Nesta seção, veremos medidas de desempenho e preço-desempenho nos sistemas de desktop usando o benchmark SPECpower.

A Figura 1.18 mostra os três servidores multiprocessadores que estamos avaliando e seu preço. Para manter justa a comparação de preços, todos são servidores Dell PowerEdge. O primeiro é o PowerEdge R710, baseado no microprocessador Intel Xeon X5670, com uma frequência de clock de 2,93 GHz. Ao contrário do Intel Core i7 abordado nos Capítulos 2 a 5, que tem quatro núcleos e um cache L3 de 8MB, esse chip da Intel tem seis núcleos e um cache L3 de 12 MB, embora os próprios núcleos sejam idênticos. Nós selecionamos um sistema de dois soquetes com 12 GB de DRAM DDR3 de 1.333 MHz protegida por ECC. O próximo servidor é o PowerEdge R815, baseado no microprocessador AMD Opteron 6174. Um chip tem seis núcleos e um cache L3 de 6 MB, e roda a 2,20 GHz, mas a AMD coloca dois desses chips em um único soquete. Assim, um soquete tem 12 núcleos e dois caches L3 de 6 MB. Nosso segundo servidor tem dois soquetes com 24 núcleos e 16 GB de DRAM DDR3 de 1.333 MHz protegido por ECC, e nosso terceiro servidor (também um PowerEdge R815) tem quatro soquetes com 48 núcleos e 32 GB de DRAM. Todos estão rodando a IBM J9 JVM e o sistema operacional Microsoft Windows 2008 Server Enterprise x64 Edition.

Observe que, devido às forças do benchmarking (Seção 1.11), esses servidores são configurados de forma pouco usual. Os sistemas na Figura 1.18 têm pouca memória em relação à capacidade de computação e somente um pequeno disco de estado sólido com 50 GB. É barato adicionar núcleos se você não precisar acrescentar aumentos proporcionais em memória e armazenamento!

Em vez de rodar, estatisticamente, programas do SPEC CPU, o SPECpower usa a mais moderna pilha de software escrita em Java. Ele é baseado no SPECjbb e representa o lado do servidor das aplicações de negócios, com o desempenho medido como o número de

	Sistema 1		Sistema 2		Sistema 3	
Componente		**Custo (% Custo)**		**Custo (% Custo)**		**Custo (% Custo)**
Servidor base	PowerEdge R710	US$ 653 (7%)	PowerEdge R815	US$ 1.437 (15%)	PowerEdge R815	US$ 1.437 (11%)
Fonte de alimentação	570 W		1.100 W		1.100 W	
Processador	Xeon X5670	US$ 3.738 (40%)	Opteron 6174	US$ 2.679 (29%)	Opteron 6174	US$ 5.358 (42%)
Frequência de clock	2,93 GHz		2,20 GHz		2,20 GHz	
Total de núcleos	12		24		48	
Soquetes	2		2		4	
Núcleos/soquete	6		12		12	
DRAM	12 GB	US$ 484 (5%)	16 GB	US$ 693 (7%)	32 GB	US$ 1.386 (11%)
Ethernet Inter.	Dual 1-Gbit	US$ 199 (2%)	Dual 1-Gbit	US$ 199 (2%)	Dual 1-Gbit	US$ 199 (2%)
Disco	50 GB SSD	US$ 1.279 (14%)	50 GB SSD	US$ 1.279 (14%)	50 GB SSD	US$ 1.279 (10%)
Windows OS		US$ 2.999 (32%)		US$ 2.999 (33%)		US$ 2.999 (24%)
Total		US$ 9.352 (100%)		US$ 9.286 (100%)		US$ 12.658 (100%)
Máx ssj_ops.	910.978		926.676		1.840.450	
Máx. ssj_ops/US$	97		100		145	

FIGURA 1.18 **Três servidores Dell PowerEdge e seus preços com base em agosto de 2010.**
Nós calculamos o custo dos processadores subtraindo o custo de um segundo processador. Do mesmo modo, calculamos o custo geral da memória vendo qual seria o custo da memória extra. Portanto, o custo-base do servidor é ajustado subtraindo o custo estimado do processador e a memória-padrão. O Capítulo 5 descreve como esses sistemas multissoquetes se conectam.

transações por segundo, chamado ssj_ops para operações por segundo do lado do servidor Java. Ele utiliza não só o processador do servidor, como o SPEC CPU, mas também as caches, o sistema de memória e até mesmo o sistema de interconexão dos multiprocessadores. Além disso, utiliza a Java Virtual Machine (JVM), incluindo o compilador de runtime JIT e o coletor de lixo, além de partes do sistema operacional.

Como mostram as duas últimas linhas da Figura 1.18, o vencedor em desempenho e preço-desempenho é o PowerEdge R815, com quatro soquetes e 48 núcleos. Ele atinge 1,8 M ssj_ops, e o ssj_ops por dólar é o mais alto, com 145. Surpreendentemente, o computador com o maior número de núcleos é o mais eficiente em termos de custo. Em segundo lugar está o R815 de dois soquetes, com 24 núcleos, e o R710 com 12 núcleos em último lugar.

Enquanto a maioria dos benchmarks (e dos arquitetos de computadores) se preocupa somente com o desempenho dos sistemas com carga máxima, os computadores raramente rodam com carga máxima. De fato, a Figura 6.2, no Capítulo 6, mostra os resultados da medição, utilizando dezenas de milhares de servidores ao longo de seis meses no Google, e menos de 1% operam com uma utilização média de 100%. A maioria tem utilização média entre 10-50%. Assim, o benchmark SPECpower captura a potência conforme a carga de trabalho-alvo varia de pico em intervalos de 10% até 0%, chamado *Active Idle*.

A Figura 1.19 mostra o ssj_ops (operações SSJ/segundo) por watt, e a potência média conforme a carga-alvo varia de 100% a 0%. O Intel R710 tem sempre a menor potência e o melhor ssj_ops por watt em todos os níveis de carga de trabalho-alvo.

Uma razão é a fonte de alimentação muito maior para o R815 com 1.110 watts *versus* 570 no R715. Como o Capítulo 6 mostra, a eficiência da fonte de alimentação é muito

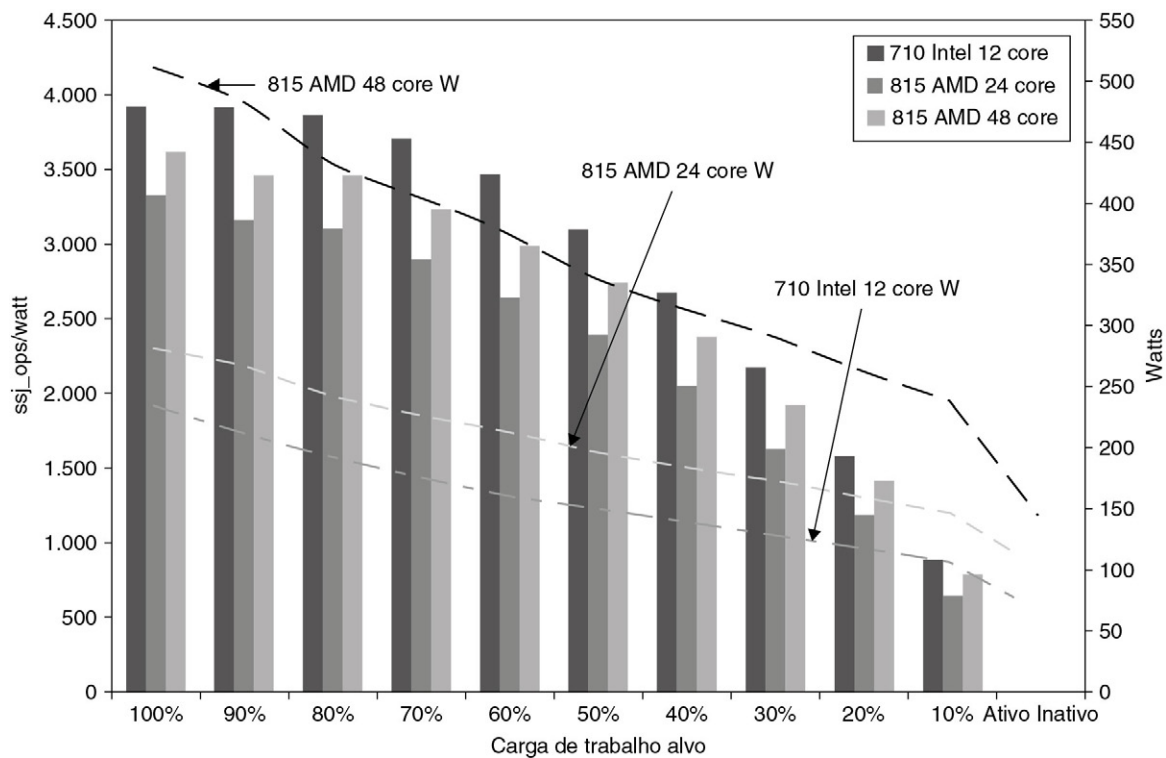



FIGURA 1.19 Desempenho de potência dos três servidores na Figura 1.18.

Os valores de ssj_ops/watt estão no eixo esquerdo, com as três colunas associadas a eles, e os valores em watts estão no eixo direito, com as três linhas associadas a eles. O eixo horizontal mostra a carga de trabalho-alvo e também consome a menor potência a cada nível.

importante na eficiência energética geral de um computador. Uma vez que watts = joules/segundo, essa métrica é proporcional às operações SSJ por joule:

$$\frac{\text{ssj_ops/s}}{\text{Watt}} = \frac{\text{ssj_ops/s}}{\text{Joule/s}} = \frac{\text{ssj_ops/s}}{\text{Joule}}$$

Para calcular um número único para comparar a eficiência energética dos sistemas, o SPECpower usa:

$$\text{ssj_ops/watt médio} = \frac{\sum \text{ssj_ops}}{\sum \text{potência}}$$

O ssj_ops/watt médio dos três servidores é de 3034 para o Intel R710, de 2357 para o AMD R815 de dois soquetes e de 2696 para o AMD R815 de quatro soquetes. Portanto, o Intel R710 tem a melhor potência/desempenho. Dividindo pelo preço dos servidores, o ssj_ops/watt/US$ 1.000 é de 324 para o Intel R710, de 254 para o AMD R815 de dois soquetes e de 213 para AMD R815 de quatro soquetes. Assim, adicionar potência reverte os resultados da competição preço-desempenho, e o troféu do preço-potência-desempenho vai para o Intel R710; o R815 de 48 núcleos vem em último lugar.

1.11 FALÁCIAS E ARMADILHAS

A finalidade desta seção, que consta de todos os capítulos, é explicar algumas das crenças erradas ou conceitos indevidos que você deverá evitar. Chamamos a esses conceitos *falácias*. Ao analisar uma falácia, tentamos oferecer um contraexemplo. Também dis-

cutimos as *armadilhas* — erros cometidos com facilidade. Essas armadilhas costumam ser generalizações de princípios que são verdadeiros em um contexto limitado. A finalidade dessas seções é ajudá-lo a evitar cometer esses erros nos computadores que você projeta.

Falácia. *Multiprocessadores são uma bala de prata.*

A mudança para múltiplos processadores por chip em meados de 2005 não veio de nenhuma descoberta que simplificou drasticamente a programação paralela ou tornou mais fácil construir computadores multicore. Ela ocorreu porque não havia outra opção, devido aos limites de ILP e de potência. Múltiplos processadores por chip não garantem potência menor. Certamente é possível projetar um chip multicore que use mais potência. O potencial que existe é o de continuar melhorando o desempenho com a substituição de um núcleo ineficiente e com alta taxa de clock por diversos núcleos eficientes e com taxa de clock mais baixa. Conforme a tecnologia melhora na redução dos transistores, isso pode encolher um pouco a capacitância e a tensão de alimentação para que possamos obter um modesto aumento no número de núcleos por geração. Por exemplo, nos últimos anos, a Intel tem adicionado dois núcleos por geração.

Como veremos nos Capítulos 4 e 5, hoje o desempenho é o fardo dos programadores. A época de o programador não levantar um só dedo e confiar nos projetistas de hardware para fazer seus programas funcionarem mais rápido está oficialmente terminada. Se os programadores quiserem que seus programas funcionem mais rápido a cada geração, deverão tornar seus programas mais paralelos.

A versão popular da lei de Moore — aumentar o desempenho a cada geração da tecnologia — está agora a cargo dos programadores.

Armadilha. *Desprezar a lei de Amdahl.*

Praticamente todo arquiteto de computadores praticante conhece a lei de Amdahl. Apesar disso, quase todos nós, uma vez ou outra, empenhamos um esforço enorme otimizando algum recurso antes de medir seu uso. Somente quando o ganho de velocidade geral é decepcionante nos lembramos de que deveríamos tê-lo medido antes de gastar tanto esforço para melhorá-lo!

Armadilha. *Um único ponto de falha.*

Os cálculos de melhoria de confiabilidade utilizando a lei de Amdahl na página 43 mostram que a dependência não é mais forte do que o elo mais fraco de uma corrente. Não importa quão mais dependente façamos as fontes de alimentação, como fizemos em nosso exemplo — o único ventilador limitará a confiabilidade do subsistema de disco. Essa observação da lei de Amdahl levou a uma regra prática para sistemas tolerantes a falhas para certificar que cada componente fosse redundante, de modo que nenhuma falha em um componente isolado pudesse parar o sistema inteiro.

Falácia. *As melhorias de hardware que aumentam o desempenho incrementam a eficiência energética ou, no pior dos casos, são neutras em termos de energia.*

Esmaelizadeh *et al.* (2011) mediram o SPEC2006 em apenas um núcleo de um Intel Core i7 de 2,67 GHz usando o modo Turbo (Seção 1.5). O desempenho aumentou por um fator de 1,07, quando a taxa de clock aumentou para 2,94 GHz (ou um fator de 1,10), mas o i7 usou um fator de 1,37 mais joules e um fator de 1,47 mais watts-hora!

Armadilha. *Benchmarks permanecem válidos indefinidamente.*

Diversos fatores influenciam a utilidade de um benchmark como previsão do desempenho real e alguns mudam com o passar do tempo. Um grande fator que influencia a utilidade

de um benchmark é a sua capacidade de resistir ao "cracking", também conhecido como "engenharia de benchmark" ou "benchmarksmanship". Quando um benchmark se torna padronizado e popular, existe uma pressão tremenda para melhorar o desempenho por otimizações direcionadas ou pela interpretação agressiva das regras para execução do benchmark. Pequenos kernels ou programas que gastam seu tempo em um número muito pequeno de linhas de código são particularmente vulneráveis.

Por exemplo, apesar das melhores intenções, o pacote de benchmark SPEC89 inicial incluía um pequeno kernel, chamado matrix300, que consistia em oito multiplicações diferentes de matrizes de 300 × 300. Nesse kernel, 99% do tempo de execução estava em uma única linha (SPEC, 1989). Quando um compilador IBM otimizava esse loop interno (usando uma ideia chamada *bloqueio*, discutida nos Capítulos 2 e 4), o desempenho melhorava por um fator de 9 em relação à versão anterior do compilador! Esse benchmark testava o ajuste do compilador e, naturalmente, não era uma boa indicação do desempenho geral nem do valor típico dessa otimização em particular.

Por um longo período, essas mudanças podem tornar obsoleto até mesmo um benchmark bem escolhido; o Gcc é o sobrevivente solitário do SPEC89. A Figura 1.16 apresenta o *status* de todos os 70 benchmarks das diversas versões SPEC. Surpreendentemente, quase 70% de todos os programas do SPEC2000 ou anteriores foram retirados da versão seguinte.

Armadilha. *O tempo médio para falha avaliado para os discos é de 1.200.000 horas ou quase 140 anos, então os discos praticamente nunca falham.*

As práticas de marketing atuais dos fabricantes de disco podem enganar os usuários. Como esse MTTF é calculado? No início do processo, os fabricantes colocarão milhares de discos em uma sala, deixarão em execução por alguns meses e contarão a quantidade que falha. Eles calculam o MTTF como o número total de horas que os discos trabalharam acumuladamente dividido pelo número daqueles que falharam.

Um problema é que esse número é muito superior ao tempo de vida útil de um disco, que normalmente é considerado cinco anos ou 43.800 horas. Para que esse MTTF grande faça algum sentido, os fabricantes de disco argumentam que o modelo corresponde a um usuário que compra um disco e depois o substitui a cada cinco anos — tempo de vida útil planejado do disco. A alegação é que, se muitos clientes (e seus bisnetos) fizerem isso no século seguinte, substituirão, em média, um disco 27 vezes antes de uma falha ou por cerca de 140 anos.

Uma medida mais útil seria a porcentagem de discos que falham. Considere 1.000 discos com um MTTF de 1.000.000 de horas e que os discos sejam usados 24 horas por dia. Se você substituir os discos que falharam por um novo com as mesmas características de confiabilidade, a quantidade que falhará em um ano (8.760 horas) será

$$\begin{aligned}\text{Discos que falham} &= \frac{\text{Número de discos} \times \text{Período de tempo}}{\text{MTTF}}\\ &= \frac{1.000 \text{ discos} \times 8.760 \text{ horas / drive}}{1.000.000 \text{ horas / falha}} = 9\end{aligned}$$

Em outras palavras, 0,9% falharia por ano ou 4,4% por um tempo de vida útil de cinco anos.

Além do mais, esses números altos são cotados com base em intervalos limitados de temperaturas e vibração; se eles forem ultrapassados, todas as apostas falharão. Um estudo recente das unidades de disco em ambientes reais (Gray e Van Ingen, 2005) afirma que cerca de 3-7% dos drives falham por ano, ou um MTTF de cerca de 125.000-

300.000 horas, e cerca de 3-7% das unidades ATA falham por ano, ou um MTTF de cerca de 125.000-300.000 horas. Um estudo ainda maior descobriu taxas de falha de disco de 2-10% (Pinheiro, Weber e Barroso, 2007). Portanto, o MTTF do mundo real é de cerca de 2-10 vezes pior que o MTTF do fabricante.

A única definição universalmente verdadeira do desempenho de pico é "o nível de desempenho que um computador certamente não ultrapassará". A Figura 1.20 mostra a porcentagem do desempenho de pico para quatro programas e quatro multiprocessadores. Ele varia de 5-58%. Como a diferença é muito grande e pode variar significativamente por benchmark, o desempenho de pico geralmente não é útil na previsão do desempenho observado.

Armadilha. *Detecção de falha pode reduzir a disponibilidade.*

Essa armadilha aparentemente irônica ocorre pelo fato de o hardware de computador possuir grande quantidade de estados que nem sempre são cruciais para determinada operação. Por exemplo, não é fatal se um erro ocorrer em uma previsão de desvio (branch), pois somente o desempenho será afetado.

Em processadores que tentam explorar agressivamente o paralelismo em nível de instrução, nem todas as operações são necessárias para a execução correta do programa. Mukherjee *et al.* (2003) descobriram que menos de 30% das operações estavam potencialmente no caminho crítico para os benchmarks SPEC2000 rodando em um Itanium 2.

A mesma observação é verdadeira sobre os programas. Se um registrador estiver "morto" em um programa — ou seja, se o programa escrever antes de ler novamente —, os erros não importarão. Se você tivesse que interromper um programa ao detectar uma falha transiente em um registrador morto, isso reduziria a disponibilidade desnecessariamente.

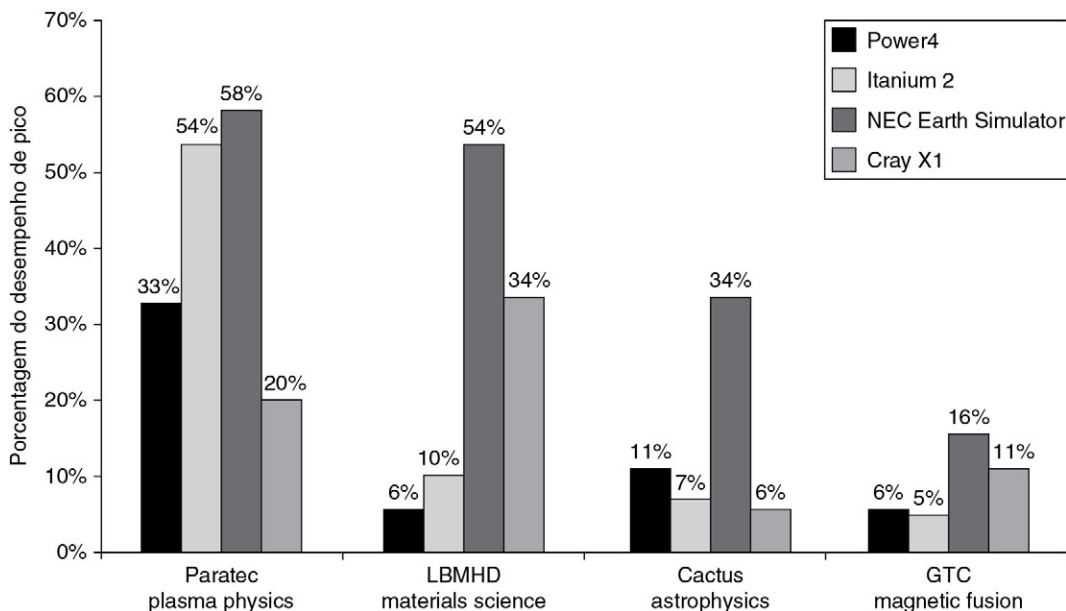



FIGURA 1.20 Porcentagem do desempenho de pico para quatro programas em quatro multiprocessadores aumentados para 64 processadores.

O Earth Simulator e o X1 são processadores de vetor (Cap. 4 e o Apêndice G). Eles não apenas oferecem uma fração mais alta do desempenho de pico, mas também têm o maior desempenho de pico e as menores taxas de clock. Exceto para o programa da Paratec, os sistemas Power 4 e Itanium 2 oferecem entre 5-10% de seu pico. *De Oliker* et al. *(2004).*

A Sun Microsystems viveu essa armadilha em 2000, com uma cache L2 que incluía paridade, mas não correção de erro, em seus sistemas Sun E3000 a Sun E10000. As SRAMs usadas para criar as caches tinham falhas intermitentes, que a paridade detectou. Se os dados na cache não fossem modificados, o processador simplesmente os relia. Como os projetistas não protegeram a cache com ECC, o sistema operacional não tinha opção a não ser informar um erro aos dados sujos e interromper o programa. Os engenheiros de campo não descobriram problemas na inspeção de mais de 90% desses casos.

Para reduzir a frequência de tais erros, a Sun modificou o sistema operacional Solaris para "varrer" a cache com um processo que escreve proativamente os dados sujos na memória. Como os chips dos processadores não tinham pinos suficientes para acrescentar ECC, a única opção de hardware para os dados sujos foi duplicar a cache externa, usando a cópia sem o erro de paridade para corrigir o erro.

A armadilha está na detecção de falhas sem oferecer um mecanismo para corrigi-las. A Sun provavelmente não disponibilizará outro computador sem ECC nas caches externas.

1.12 COMENTÁRIOS FINAIS

Este capítulo introduziu uma série de conceitos e forneceu um framework quantitativo que expandiremos ao longo do livro. A partir desta edição, a eficiência energética é a nova companheira do desempenho.

No Capítulo 2, iniciaremos a importantíssima área do projeto de sistema de memória. Vamos examinar uma vasta gama de técnicas que conspiram para fazer a memória parecer infinitamente grande e, ainda assim, o mais rápida possível (o Apêndice B fornece material introdutório sobre caches para leitores sem formação nem muita experiência nisso). Como nos capítulos mais adiante, veremos que a cooperação entre hardware e software se tornou essencial para os sistemas de memória de alto desempenho, assim como para os pipelines de alto desempenho. Este capítulo também aborda as máquinas virtuais, uma técnica de proteção cada vez mais importante.

No Capítulo 3, analisaremos o paralelismo em nível de instrução (Instruction-Level Parallelism — ILP), cuja forma mais simples e mais comum é o pipelining. A exploração do ILP é uma das técnicas mais importantes para a criação de uniprocessadores de alta velocidade. A presença de dois capítulos reflete o fato de que existem várias técnicas para a exploração do ILP e que essa é uma tecnologia importante e amadurecida. O Capítulo 3 começa com uma longa apresentação dos conceitos básicos que o prepararão para a grande gama de ideias examinadas nos dois capítulos anteriores. Ele utiliza exemplos disseminados há cerca de 40 anos, abarcando desde um dos primeiros supercomputadores (IBM 360/91) até os processadores mais rápidos do mercado em 2011. Além disso, enfatiza a técnica para a exploração do ILP, chamada *dinâmica* ou *em tempo de execução*. Também focaliza os limites e as extensões das ideias do ILP e apresenta o multithreading, que será detalhado nos Capítulos 4 e 5. O Apêndice C é um material introdutório sobre pipelining para os leitores sem formação ou muita experiência nesse assunto. (Esperamos que ele funcione como uma revisão para muitos leitores, incluindo os do nosso texto introdutório, *Computer Organization and Design: The Hardware/Software Interface*.)

O Capítulo 4 foi escrito para esta edição e explica três modos de explorar o paralelismo em nível de dados. A abordagem clássica, mais antiga, é a arquitetura vetorial, e começamos por ela para estabelecer os princípios do projeto SIMD (o Apêndice G apresenta detalhes sobre as arquiteturas vetoriais). Em seguida, explicamos as extensões de conjunto de instruções encontradas na maioria dos microprocessadores desktops atuais. A terceira parte

é uma explicação aprofundada de como as unidades de processamento gráfico (GPUs) modernas funcionam. A maioria das descrições de GPU é feita da perspectiva do programador, que geralmente oculta o modo como o computador realmente funciona. Essa seção explica as GPUs da perspectiva de alguém "de dentro", incluindo um mapeamento entre os jargões de GPU e os termos de arquitetura mais tradicionais.

O Capítulo 5 enfoca como obter alto desempenho usando múltiplos processadores ou multiprocessadores. Em vez de usar o paralelismo para sobrepor instruções individuais, o multiprocessamento o utiliza para permitir que vários fluxos de instruções sejam executados simultaneamente em diferentes processadores. Nosso foco recai sobre a forma dominante dos multiprocessadores, os multiprocessadores de memória compartilhada, embora também apresentemos outros tipos e discutamos os aspectos mais amplos que surgem em qualquer multiprocessador. Aqui, mais uma vez, exploramos diversas técnicas, focalizando as ideias importantes apresentadas inicialmente nas décadas de 1980 e 1990.

O Capítulo 6 também foi criado para esta edição. Apresentamos os clusters e, depois, tratamos detalhadamente dos computadores de escala warehouse (warehouse-scale computers — WSCs) que os arquitetos de computadores ajudam a projetar. Os projetistas de WSCs são os descendentes profissionais dos pioneiros dos supercomputadores, como Seymour Cray, pois vêm projetando computadores extremos. Eles contêm dezenas de milhares de servidores, e seu equipamento e sua estrutura custam cerca de US$ 200 milhões. Os problemas de preço-desempenho e eficiência energética abordados nos primeiros capítulos aplicam-se aos WSCs, assim como a abordagem quantitativa de tomada de decisões.

Este livro conta com grande quantidade de material on-line (ver mais detalhes no Prefácio), tanto para reduzir o custo quanto para apresentar aos leitores diversos tópicos avançados. A Figura 1.21 apresenta todo esse material. Os Apêndices A, B e C, incluídos neste volume, funcionarão como uma revisão para muitos leitores.

No Apêndice D, nos desviaremos de uma visão centrada no processador e examinaremos questões sobre sistemas de armazenamento. Aplicamos um enfoque quantitativo semelhante, porém baseado em observações do comportamento do sistema, usando uma técnica de ponta a ponta para a análise do desempenho. Ele focaliza a importante questão de como armazenar e recuperar dados de modo eficiente usando principalmente

Apêndice	Título
A	Princípios de conjuntos de instruções
B	Revisão de hierarquia de memória
C	Pipelining: conceitos básicos e intermediários
D	Sistemas de armazenamento
E	Sistemas embarcados
F	Redes de interconexão
G	Processadores vetoriais em mais detalhes
H	Hardware e software para VLIW e EPIC
I	Multiprocessadores em grande escala e aplicações científicas
J	Aritmética de computador
K	Inspeção das arquiteturas de conjunto de instruções
L	Perspectivas históricas e referências

FIGURA 1.21 Lista de apêndices.

as tecnologias de armazenamento magnético de menor custo. Nosso foco recai sobre o exame do desempenho dos sistemas de armazenamento de disco para cargas de trabalho típicas de E/S, como os benchmarks OLTP que vimos neste capítulo. Exploramos bastante os tópicos avançados nos sistemas baseados em RAID, que usam discos redundantes para obter alto desempenho e alta disponibilidade. Finalmente, o capítulo apresenta a teoria de enfileiramento, que oferece uma base para negociar a utilização e a latência.

O Apêndice E utiliza um ponto de vista de computação embarcada para as ideias de cada um dos capítulos e apêndices anteriores.

O Apêndice F explora o tópico de interconexão de sistemas mais abertamente, incluindo WANs e SANs, usadas para permitir a comunicação entre computadores.

Ele também descreve os clusters, que estão crescendo em importância, devido à sua adequação e eficiência para aplicações de banco de dados e servidor Web.

O Apêndice H revê hardware e software VLIW, que, por contraste, são menos populares do que quando o EPIC apareceu em cena, um pouco antes da última edição.

O Apêndice I descreve os multiprocessadores em grande escala para uso na computação de alto desempenho.

O Apêndice J é o único que permanece desde a primeira edição. Ele abrange aritmética de computador.

O Apêndice K é um estudo das arquiteturas de instrução, incluindo o 80x86, o IBM 360, o VAX e muitas arquiteturas RISC, como ARM, MIPS, Power e SPARC.

Descreveremos o Apêndice L mais adiante.

1.13 PERSPECTIVAS HISTÓRICAS E REFERÊNCIAS

O Apêndice L (disponível on-line) inclui perspectivas históricas sobre as principais ideias apresentadas em cada um dos capítulos deste livro. Essas seções de perspectiva histórica nos permitem rastrear o desenvolvimento de uma ideia por uma série de máquinas ou descrever projetos significativos. Se você estiver interessado em examinar o desenvolvimento inicial de uma ideia ou máquina, ou em ler mais sobre o assunto, são dadas referências ao final de cada história. Sobre este capítulo, consulte a Seção L.2, “O desenvolvimento inicial dos computadores”, para obter uma análise do desenvolvimento inicial dos computadores digitais e das metodologias de medição de desempenho.

Ao ler o material histórico, você logo notará que uma das maiores vantagens da juventude da computação, em comparação com vários outros campos da engenharia, é que muitos dos pioneiros ainda estão vivos — podemos aprender a história simplesmente perguntando a eles!

ESTUDOS DE CASO E EXERCÍCIOS POR DIANA FRANKLIN

Estudo de caso 1: custo de fabricação de chip

Conceitos ilustrados por este estudo de caso

- Custo de fabricação
- Rendimento da fabricação
- Tolerância a defeitos pela redundância

Existem muitos fatores envolvidos no preço de um chip de computador. Tecnologia nova e menor oferece aumento no desempenho e uma queda na área exigida para o chip. Na tecno-

Chip	Tamanho do die (mm^2)	Taxa de defeito estimada (por cm^2)	Tamanho de manufatura (nm)	Transistores (milhões)
IBM Power5	389	0,3	130	276
Sun Niagara	380	0,75	90	279
AMD Opteron	199	0,75	90	233

FIGURA 1.22 Fatores de custo de manufatura para vários processadores modernos.

logia menor, pode-se manter a área pequena ou colocar mais hardware no chip, a fim de obter mais funcionalidade. Neste estudo de caso, exploramos como diferentes decisões de projeto envolvendo tecnologia de fabricação, superfície e redundância afetam o custo dos chips.

1.1 [10/10] <1.6> A Figura 1.22 contém uma estatística relevante de chip, que influencia o custo de vários chips atuais. Nos próximos exercícios, você vai explorar as escolhas envolvidas para o IBM Power5.

a. [10] <1.6> Qual é o rendimento para o IBM Power5?

b. [10] <1.6> Por que o IBM Power5 tem uma taxa de defeitos pior do que o Niagara e o Opteron?

1.2 [20/20/20/20] <1,6> Custa US$ 1 bilhão montar uma nova instalação de fabricação. Você vai produzir diversos chips nessa fábrica e precisa decidir quanta capacidade dedicar a cada chip. Seu chip Woods terá uma área de 150 mm^2, vai lucrar US$ 20 por chip livre de defeitos. Seu chip Markon terá 250 mm^2 e vai gerar um lucro de US$ 225 por chip livre de defeitos. Sua instalação de fabricação será idêntica àquela do Power5. Cada wafer tem 300 mm de diâmetro.

a. [20] <1.6> Quanto lucro você obterá com cada wafer do chip Woods?

b. [20] <1.6> Quanto lucro você obterá com cada wafer do chip Markon?

c. [20] <1.6> Que chip você deveria produzir nessa instalação?

d. [20] <1.6> Qual é o lucro em cada novo chip Power5? Se sua demanda é de 50.000 chips Woods por mês e 25.000 chips Markon por mês, e sua instalação pode fabricar 150 wafers em um mês, quantos wafers de cada chip você deveria produzir?

1.3 [20/20] <1.6> Seu colega na AMD sugere que, já que o rendimento é tão pobre, você poderia fabricar chips mais rapidamente se colocasse um núcleo extra no die e descartasse somente chips nos quais os dois processadores tivessem falhado. Vamos resolver este exercício vendo o rendimento como a probabilidade de não ocorrer nenhum defeito em certa área, dada a taxa de defeitos. Calcule as probabilidades com base em cada núcleo Opteron separadamente (isso pode não ser inteiramente preciso, já que a equação do rendimento é baseada em evidências empíricas, e não em um cálculo matemático relacionando as probabilidades de encontrar erros em partes diferentes do chip).

a. [20] <1.6> Qual é a probabilidade de um defeito ocorrer em somente um dos núcleos?

b. [10] <1.6> Se o chip antigo custar US$ 20 por unidade, qual será o custo do novo chip, levando em conta a nova área e o rendimento?

Estudo de caso 2: consumo de potência nos sistemas de computador

Conceitos ilustrados por este estudo de caso

- Lei de Amdahl
- Redundância
- MTTF
- Consumo de potência

O consumo de potência nos sistemas modernos depende de uma série de fatores, incluindo a frequência de clock do chip, a eficiência, a velocidade da unidade de disco, a utilização da unidade de disco e a DRAM. Os exercícios a seguir exploram o impacto sobre a potência que tem diferentes decisões de projeto e/ou cenários de uso.

1.4 [20/10/20] <1.5> A Figura 1.23 apresenta o consumo de potência de vários componentes do sistema de computador. Neste exercício, exploraremos como o disco rígido afeta o consumo de energia para o sistema.

Tipo de componente	Produto	Desempenho	Potência
Processador	Sun Niagara 8-Core	1,2 GHz	72-79 W pico
	Intel Pentium 4	2 GHz	48,9-66 W
DRAM	Kingston X64C3AD2 1 GB	184 pinos	3,7 W
	Kingston D2N3 1 GB	240 pinos	2,3 W
Disco rígido	DiamondMax 16	5.400 rpm	7,0 W leitura/busca, 2,9 W ocioso
	DiamondMax Plus 9	7.200 rpm	7,9 W leitura/busca, 4,0 W ocioso

FIGURA 1.23 Consumo de potência de vários componentes do computador.

- **a.** [20] <1.5> Considerando a carga máxima para cada componente e uma eficiência da fonte de alimentação de 80%, que potência, em watts, a fonte de alimentação do servidor precisa fornecer a um sistema com um chip Intel Pentium 4 com DRAM Kingston de 2 GB e 240 pinos, e duas unidades de disco rígido de 7.200 rpm?
- **b.** [10] <1.5> Quanta potência a unidade de disco de 7.200 rpm consumirá se estiver ociosa aproximadamente 60% do tempo?
- **c.** [20] <1.5> Dado que o tempo de leitura de dados de um drive de disco de 7.200 rpm será aproximadamente 75% do de um disco de 5.400 rpm, com qual tempo de inatividade do disco de 7.200 rpm o consumo de energia será igual, na média, para os dois discos?

1.5 [10/10/20] <1.5> Um fator crítico no cálculo de potência de um conjunto de servidores é o resfriamento. Se o calor não for removido do computador com eficiência, os ventiladores devolverão ar quente ao computador em vez de ar frio. Veremos como diferentes decisões de projeto afetam o resfriamento necessário e, portanto, o preço de um sistema. Use a Figura 1.23 para fazer os cálculos de potência.

- **a.** [10] <1.5> Uma porta de resfriamento para um rack custa US$ 4.000 e dissipa 14 KW (na sala; um custo adicional é necessário para que saia da sala). Quantos servidores com processador Intel Pentium 4, DRAM de 1 GB em 240 pinos e um único disco rígido de 7.200 rpm você pode resfriar com uma porta de resfriamento?
- **b.** [10] <1.5> Você está considerando o fornecimento de tolerância a falhas para a sua unidade de disco rígido. O RAID 1 dobra o número de discos (Cap. 6). Agora, quantos sistemas você pode colocar em um único rack com um único cooler?
- **c.** [20] <1.5> Conjunto de servidores típicos pode dissipar no máximo 200 W por pé quadrado. Dado que um rack de servidor requer 11 pés quadrados (incluindo espaços na frente e atrás), quantos servidores da parte (a) podem se colocados em um único rack e quantas portas de resfriamento são necessárias?

1.6 [Discussão] <1.8> A Figura 1.24 oferece uma comparação da potência e do desempenho para vários benchmarks considerando dois servidores: Sun Fire T2000 (que usa o Niagara) e IBM x346 (que usa processadores Intel Xeon). Essa

	Sun Fire T2000	IBM x346
Potência (watts)	298	438
SPECjbb (op/s)	63.378	39.985
Potência (watts)	330	438
SPECWeb (composto)	14.001	4.348

FIGURA 1.24 Comparação de potência/desempenho do Sun, conforme informado seletivamente pela Sun.

informação foi reportada em um site da Sun. Existem duas informações reportadas: potência e velocidade em dois benchmarks. Para os resultados mostrados, o Sun Fire T2000 é claramente superior. Que outros fatores poderiam ser importantes a ponto de fazer alguém escolher o IBM x346 se ele fosse superior nessas áreas?

1.7 [20/20/20/20] <1.6, 1.9> Os estudos internos da sua empresa mostram que um sistema de único núcleo é suficiente para a demanda na sua capacidade de processamento. Porém, você está pesquisando se poderia economizar potência usando dois núcleos.

a. [20] <1.9> Suponha que sua aplicação seja 80% paralelizável. Por quanto você poderia diminuir a frequência e obter o mesmo desempenho?

b. [20] <1.6> Considere que a voltagem pode ser diminuída linearmente com a frequência. Usando a equação na Seção 1.5, quanta potência dinâmica o sistema de dois núcleos exigiria em comparação com o sistema de único núcleo?

c. [20] <1.6, 1.9> Agora considere que a tensão não pode cair para menos de 25% da voltagem original. Essa tensão é conhecida como "piso de tensão", e qualquer voltagem inferior a isso perderá o estado. Que porcentagem de paralelização lhe oferece uma tensão no piso de tensão?

d. [20] <1.6, 1.9> Usando a equação da Seção 1.5, quanta potência dinâmica o sistema de dois núcleos exigiria em comparação com o sistema de único núcleo, levando em consideração o piso de tensão?

Exercícios

1.8 [10/15/15/10/10] <1.1,1.5> Um desafio para os arquitetos é que o projeto criado hoje vai requerer muitos anos de implementação, verificação e testes antes de aparecer no mercado. Isso significa que o arquiteto deve projetar, muitos anos antes, o que a tecnologia será. Às vezes, isso é difícil de fazer.

a. [10] <1.4> De acordo com a tendência em escala de dispositivo observada pela lei de Moore, o número de transistores em 2015 será quantas vezes o número de transistores em 2005?

b. [10] <1.5> Um dia, o aumento nas frequências de clock acompanhou essa tendência. Se as frequências de clock tivessem continuado a crescer na mesma taxa que nos anos 1990, aproximadamente quão rápidas seriam as frequências de clock em 2015?

c. [15] <1.5> Na taxa de aumento atual, quais são as frequências de clock projetadas para 2015?

d. [10] <1.4> O que limitou a taxa de aumento da frequência de clock e o que os arquitetos estão fazendo com os transistores adicionais para aumentar o desempenho?

e. [10] <1.4> A taxa de crescimento para a capacidade da DRAM também diminuiu. Por 20 anos, a capacidade da DRAM aumentou em 60% por ano. Essa taxa caiu para 40% por ano e hoje o aumento é de 25-40% por ano.

Se essa tendência continuar, qual será a taxa de crescimento aproximada para a capacidade da DRAM em 2020?

1.9 [10/10] <1.5> Você está projetando um sistema para uma aplicação em tempo real na qual prazos específicos devem ser atendidos. Terminar o processamento mais rápido não traz nenhum benefício. Você descobre que, na pior das hipóteses, seu sistema pode executar o código necessário duas vezes mais rápido do que o necessário.

- **a.** [10] <1.5> Quanta energia você economizará se executar na velocidade atual e desligar o sistema quando o processamento estiver completo?
- **b.** [10] <1.5> Quanta energia você economizará se configurar a voltagem e a frequência para a metade das atuais?

1.10 [10/10/20/10] <1.5> Conjunto de servidores, como as do Google e do Yahoo!, fornece capacidade computacional suficiente para a maior taxa de requisições do dia. Suponha que, na maior parte do tempo, esses servidores operem a 60% da capacidade. Suponha também que a potência não aumente linearmente com a carga, ou seja, quando os servidores estão operando a 60% de capacidade, consomem 90% da potência máxima. Os servidores poderiam ser desligados, mas levariam muito tempo para serem reiniciados em resposta a mais carga. Foi proposto um novo sistema, que permite um reinício rápido, mas requer 20% da potência máxima durante esse estado "quase vivo".

- **a.** [10] <1.5> Quanta economia de energia seria obtida desligando 60% dos servidores?
- **b.** [10] <1.5> Quanta economia de energia seria obtida colocando 60% dos servidores no estado "quase vivo"?
- **c.** [20] <1.5> Quanta economia de energia seria obtida reduzindo a tensão em 20% e a frequência em 40%?
- **d.** [20] <1.5> Quanta economia de energia seria obtida colocando 30% dos servidores no estado "quase vivo" e desligando 30%?

1.11 [10/10/20] <1.7> Disponibilidade é a consideração mais importante para o projeto de servidores, seguida de perto pela escalabilidade e pelo throughput.

- **a.** [10] <1.7> Temos um único processador com falhas no tempo (FIT) de 100. Qual é o tempo médio para a falha (MTTF) desse sistema?
- **b.** [10] <1.7> Se levar um dia para fazer o sistema funcionar de novo, qual será a disponibilidade desse sistema?
- **c.** [20] <1.7> Imagine que, para reduzir custos, o governo vai construir um supercomputador a partir de computadores baratos em vez de computadores caros e confiáveis. Qual é o MTTF para um sistema com 1.000 processadores? Suponha que, se um falhar, todos eles falharão.

1.12 [20/20/20] <1.1, 1.2, 1.7> Em conjunto de servidores como os usadas pela Amazon e pelo eBay, uma única falha não faz com que todo o sistema deixe de funcionar. Em vez disso, ela vai reduzir o número de requisições que podem ser satisfeitas em dado momento.

- **a.** [20] <1.7> Se uma companhia tem 10.000 computadores, cada qual com um MTTF de 35 dias, e sofre uma falha catastrófica somente quando 1/3 dos computadores falham, qual é o MTTF do sistema?
- **b.** [20] <1.1, 1.7> Se uma companhia tivesse US$ 1.000 adicionais, por computador, para dobrar o MTTF, essa seria uma boa decisão de negócio? Mostre seu trabalho.
- **c.** [20] <1.2> A Figura 1.3 mostra a média dos custos dos tempos de paralisação, supondo que o custo é igual durante o ano todo. Para os varejistas, entretanto, a época de Natal é a mais lucrativa (e, portanto, a mais prejudicada pela perda de vendas). Se um centro de vendas por catálogo tiver duas vezes mais tráfego

no quarto trimestre do que em qualquer outro, qual será o custo médio do tempo de paralisação por hora no quarto trimestre e no restante do ano?

1.13 [10/20/20] <1.9> Suponha que sua empresa esteja tentando decidir entre adquirir o Opteron e adquirir o Itanium 2. Você analisou as aplicações da sua empresa e notou que em 60% do tempo ela estará executando aplicações similares ao wupwise, em 20% do tempo aplicações similares ao ammp e em 20% do tempo aplicações similares ao apsi.

a. [10] Se você estivesse escolhendo somente com base no desempenho SPEC geral, qual seria a escolha e por quê?

b. [20] Qual é a média ponderada das taxas de tempo de execução para esse *mix* de aplicações para o Opteron e o Itanium 2?

c. [20] Qual é o ganho de velocidade do Opteron sobre o Itanium 2?

1.14 [20/10/10/10/15] <1.9> Neste exercício, suponha que estejamos considerando melhorar uma máquina adicionando a ela hardware vetorial. Quando um processamento é executado em modo vetor nesse hardware, é 10 vezes mais rápido do que o modo original de execução. Chamamos *porcentagem de vetorização* a porcentagem de tempo que seria gasta usando o modo vetorial. Vetores serão discutidos no Capítulo 4, mas você não precisa saber nada sobre como eles funcionam para responder a esta questão!

a. [20] <1.9> Trace um gráfico que plote o ganho de velocidade como uma porcentagem do processamento realizada em modo vetor. Chame o eixo *y* de "Ganho médio de velocidade" e o eixo *x* de "Porcentagem de vetorização".

b. [10] <1.9> Que porcentagem de vetorização é necessária para atingir um ganho de velocidade de 2?

c. [10] <1.9> Que porcentagem do tempo de execução do processamento será gasto no modo vetorial se um ganho de velocidade de 2 for alcançado?

d. [10] <1.9> Que porcentagem de vetorização é necessária para atingir metade do ganho de velocidade que pode ser obtido usando o modo vetorial?

e. [15] <1.9> Suponha que você tenha descoberto que a porcentagem de vetorização do programa é de 70%. O grupo de projeto de hardware estima que pode acelerar ainda mais o hardware vetorial com significativo investimento adicional. Você imagina, em vez disso, que a equipe do compilador poderia aumentar a porcentagem de vetorização. Que porcentagem de vetorização a equipe do compilador precisa atingir para igualar uma adição de 2x no ganho de velocidade na unidade vetorial (além dos 10x iniciais)?

1.15 [15/10] <1.9> Suponha que tenha sido feita uma melhoria em um computador que aumente algum modo de execução por um fator de 10. O modo melhorado é usado em 50% do tempo e medido como uma porcentagem do tempo de execução *quando o modo melhorado está em uso.* Lembre-se de que a lei de Amdahl depende da fração de tempo de execução original e *não melhorado* que poderia fazer uso do modo melhorado. Assim, não podemos usar diretamente essa medida de 50% para calcular o ganho de velocidade com a lei de Amdahl.

a. [15] <1.9> Qual é o ganho de velocidade que obtemos do modo rápido?

b. [10] <1.9> Que porcentagem do tempo de execução original foi convertida para o modo rápido?

1.16 [20/20/15] <1.9> Muitas vezes, ao fazermos modificações para otimizar parte de um processador, o ganho de velocidade em um tipo de instrução ocorre à custa de reduzir a velocidade de algo mais. Por exemplo, se adicionarmos uma complicada unidade de ponto flutuante que ocupe espaço e algo tiver de ser afastado do centro para acomodá-la, adicionar um ciclo extra de atraso para atingir essa unidade. A equação básica da lei de Amdahl não leva em conta essa troca.

a. [20] <1.9> Se a nova unidade rápida de ponto flutuante acelerar as operações do ponto flutuante numa média de 2x e as operações de ponto flutuante ocuparem 20% do tempo de execução do programa original, qual será o ganho geral de velocidade (ignorando a desvantagem de quaisquer outras instruções)?

b. [20] <1.9> Agora suponha que a aceleração da unidade de ponto flutuante reduziu a velocidade dos acessos à cache de dados, resultando em uma redução de velocidade de 1,5x (ou em ganho de velocidade de 2/3). Os acessos à cache de dados consomem 10% do tempo de execução. Qual é o ganho geral de velocidade agora?

c. [15] <1.9> Depois de implementar as novas operações de ponto flutuante, que porcentagem do tempo de execução é gasto em operações desse tipo? Que porcentagem é gasta em acessos à cache de dados?

1.17 [10/10/20/20] <1.10> Sua empresa acabou de comprar um novo processador Intel Core i5 e você foi encarregado de otimizar seu software para esse processador. Você executará duas aplicações nesse Pentium dual, mas os requisitos de recursos não são iguais. A primeira aplicação precisa de 80% dos recursos e a outra de apenas 20% dos recursos. Suponha que, quando você paraleliza uma parte do programa, o ganho de velocidade para essa parte seja de 2.

a. [10] <1.10> Se 40% da primeira aplicação fosse paralelizável, quanto ganho de velocidade você obteria com ela se fosse executada isoladamente?

b. [10] <1.10> Se 99% da segunda aplicação fosse paralelizável, quanto ganho de velocidade ela observaria se fosse executada isoladamente?

c. [20] <1.10> Se 40% da primeira aplicação fosse paralelizável, quanto *ganho de velocidade geral do sistema* você observaria se a paralelizasse?

d. [20] <1.10> Se 99% da segunda aplicação fosse paralelizável, quanto ganho de velocidade geral do sistema você obteria?

1.18 [10/20/20/20/25] <1.10> Ao paralelizar uma aplicação, o ganho de velocidade ideal é feito pelo número de processadores. Isso é limitado por duas coisas: a porcentagem da aplicação que pode ser paralelizada e o custo da comunicação. A lei de Amdahl leva em conta a primeira, mas não a segunda.

a. [10] <1.10> Qual será o ganho de velocidade com N processadores se 80% da aplicação puder ser paralelizada, ignorando o custo de comunicação?

b. [20] <1.10> Qual será o ganho de velocidade com oito processadores se, para cada processador adicionado, o custo adicional de comunicação for de 0,5% do tempo de execução original?

c. [20] <1.10> Qual será o ganho de velocidade com oito processadores se, cada vez que o número de processadores for dobrado, o custo adicional de comunicação for aumentado em 0,5% do tempo de execução original?

d. [20] <1.10> Qual será o ganho de velocidade com N processadores se, cada vez que o número de processadores for dobrado, o custo adicional de comunicação for aumentado em 0,5% do tempo de execução original?

e. [25] <1.10> Escreva a equação geral que resolva esta questão: qual é o número de processadores com o maior ganho de velocidade em uma aplicação na qual $P\%$ do tempo de execução original é paralelizável e, para cada vez que o número de processadores for dobrado, a comunicação será aumentada em 0,5% do tempo de execução original?

Projeto de hierarquia de memória

O ideal seria uma capacidade de memória indefinidamente grande, de modo que qualquer palavra em particular [...] pudesse estar imediatamente disponível [...] Somos [...] forçados a reconhecer a possibilidade de construir uma hierarquia de memórias, cada qual com maior capacidade que a anterior, porém com acesso mais lento que a outra.

A. W. Burks, H. H. Goldstine e J. von Neumann, *Preliminary Discussion of the Logical Design of na Eletronic Computing Instrument* (1946)

2.1 INTRODUÇÃO

Os pioneiros do computador previram corretamente que os programadores desejariam uma quantidade ilimitada de memória rápida. Uma solução econômica para isso é a *hierarquia de memória*, que tira proveito da localidade e da relação custo-desempenho das tecnologias de memória. O *princípio da localidade*, apresentado no Capítulo 1, afirma que a maioria dos programas não acessa todo o código ou dados uniformemente. A localidade ocorre no tempo (localidade *temporal*) e no espaço (localidade *espacial*). Esse princípio, junto com a noção de que um hardware menor pode se tornar mais rápido, levou às hierarquias baseadas em memórias de diferentes velocidades e tamanhos. A Figura 2.1 mostra uma hierarquia de memória multinível, incluindo os valores típicos do tamanho e da velocidade de acesso.

Como a memória rápida também é cara, uma hierarquia de memória é organizada em vários níveis — cada qual menor, mais rápido e mais caro por byte do que o nível inferior seguinte. O objetivo é oferecer um sistema de memória com custo por unidade quase tão baixo quanto o nível de memória mais barato e velocidade quase tão rápida quanto o nível mais rápido. Na maioria dos casos (mas nem sempre), os dados contidos em um nível inferior são um subconjunto do nível superior seguinte. Essa propriedade, chamada

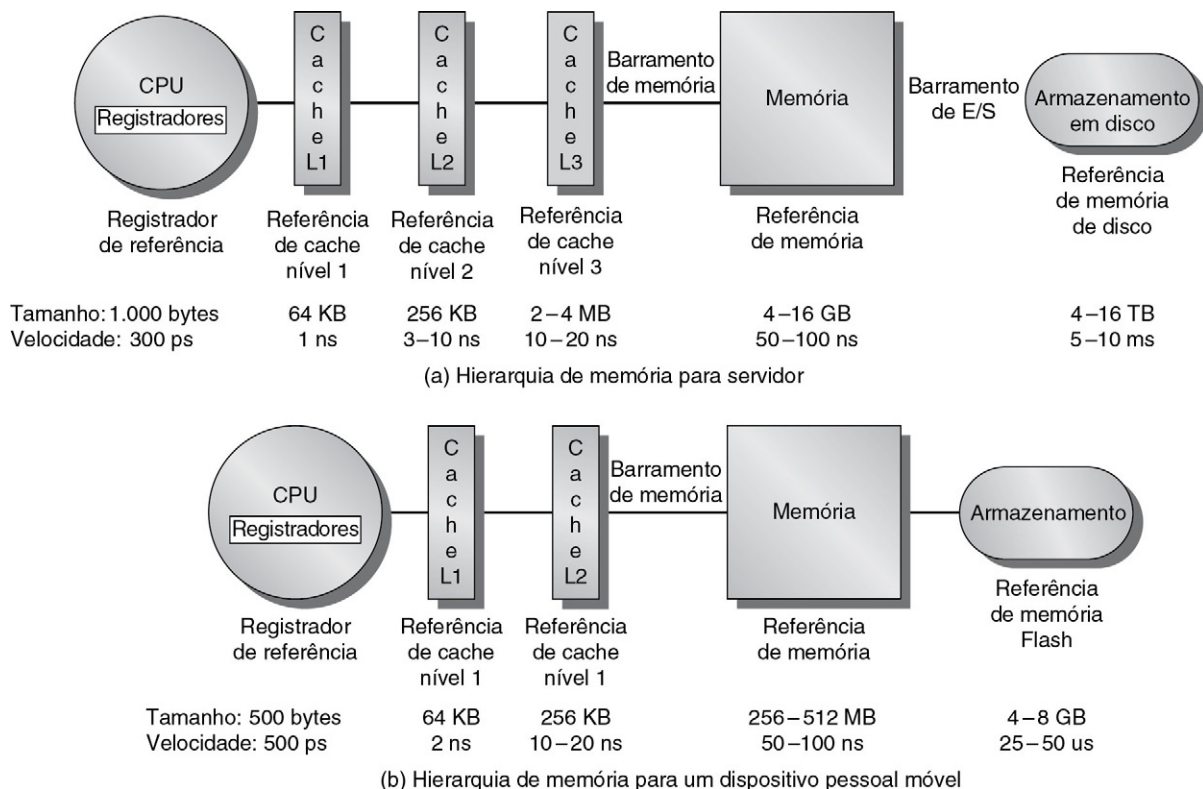



FIGURA 2.1 **Os níveis em uma hierarquia de memória em um servidor (a) e em um dispositivo pessoal móvel (PMD) (b).**
À medida que nos distanciamos do processador, a memória no nível abaixo se torna mais lenta e maior. Observe que as unidades de tempo mudam por fatores de 10^9 — de picossegundos para milissegundos — e que as unidades de tamanho mudam por fatores de 10^{12} — de bytes para terabytes. O PMD tem uma taxa de clock menor e as caches e a memória principal menores. Uma diferença-chave é que os servidores e desktops usam armazenamento de disco como o nível mais baixo na hierarquia, enquanto os PMDs usam memória Flash, construída a partir de tecnologia EEPROM.

propriedade de inclusão, é sempre necessária para o nível mais baixo da hierarquia, que consiste na memória principal, no caso das caches, e na memória de disco, no caso da memória virtual.

A importância da hierarquia de memória aumentou com os avanços no desempenho dos processadores. A Figura 2.2 ilustra as projeções do desempenho do processador contra a melhoria histórica de desempenho no tempo para acessar a memória principal. A linha do processador mostra o aumento na média das requisições de memória por segundo (ou seja, o inverso da latência entre referências de memória), enquanto a linha da memória mostra o aumento nos acessos por segundo à DRAM (ou seja, o inverso da latência de acesso à DRAM). Na verdade, a situação em um uniprocessador é um pouco pior, uma vez que o pico na taxa de acesso à memória é mais rápido do que a taxa média, que é o que é mostrado.

Mais recentemente, processadores de alto nível progrediram para múltiplos núcleos, aumentando mais os requisitos de largura de banda em comparação com os núcleos únicos. De fato, o pico de largura de banda agregada essencialmente aumenta conforme o número de núcleos aumenta. Um processador de alto nível moderno, como o Intel Core i7, pode gerar duas referências de memória de dados por núcleo a cada ciclo de clock. Com quatro núcleos em uma taxa de clock de 3,2 GHz, o i7 pode gerar um pico de 25,6 bilhões de referências a dados de 64 bits por segundo, além de um pico de demanda de instruções de cerca de 12,8 bilhões de referências a instruções de 128 bits. Isso é um pico

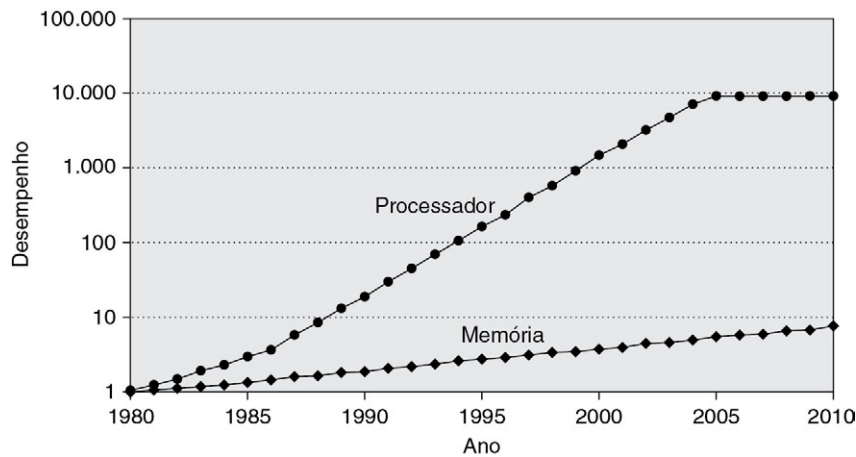



FIGURA 2.2 **Começando com o desempenho de 1980 como uma linha base, a distância do desempenho, medida como a diferença entre os requisitos da memória dos processadores (para um uniprocessador ou para um core) e a latência de um acesso à DRAM, é desenhada contra o tempo.**
Observe que o eixo vertical precisa estar em uma escala logarítmica para registrar o tamanho da diferença de desempenho processador-DRAM. A linha-base da memória é de 64 KB de DRAM em 1980, com uma melhoria de desempenho de 1,07 por ano na latência (Fig. 2.13, na página 85). A linha do processador pressupõe uma melhoria de 1,25 por ano até 1986, uma melhoria de 1,52 até 2000, uma melhoria de 1,20 entre 2000 e 2005, e nenhuma mudança no desempenho do processador (tendo por base um núcleo por core) entre 2005 e 2010 (Fig. 1.1, no Cap. 1).

de largura de banda total de 409,6 GB/s! Essa incrível largura de banda é alcançada pelo multiporting e pelo pipelining das caches; pelo uso de níveis múltiplos de caches, usando caches de primeiro — e às vezes segundo — nível separados por núcleo; e pelo uso de caches de dados e instruções separados no primeiro nível. Em contraste, o pico de largura de banda para a memória principal DRAM é de somente 6% desse valor (25 GB/s).

Tradicionalmente, os projetistas de hierarquias de memória se concentraram em otimizar o tempo médio de acesso à memória, que é determinado pelo tempo de acesso à cache, taxa de falta e penalidade por falta. Mais recentemente, entretanto, a potência tornou-se uma importante consideração. Em microprocessadores de alto nível, pode haver 10 MB ou mais de cache no chip, e uma grande cache de segundo — ou terceiro — nível vai consumir potência significativa, tanto como fuga, quando ele não está operando (chamada *potência estática*) quanto como potência ativa quando uma leitura ou gravação é realizada (chamada *potência dinâmica*), como descrito na Seção 2.3. O problema é ainda mais sério em processadores em PMDs, nos quais a CPU é menos agressiva e a necessidade de potência pode ser 20-50 vezes menor. Nesses casos, as caches podem ser responsáveis por 25-50% do consumo total de potência. Assim, mais projetos devem considerar a relação de desempenho e da potência, que serão examinados neste capítulo.

O básico das hierarquias de memória: uma revisão rápida

O tamanho crescente e, portanto, a importância dessa diferença levou à migração dos fundamentos de hierarquia de memória para os cursos de graduação em arquitetura de computador e até mesmo para cursos de sistemas operacionais e compiladores. Assim, começaremos com uma rápida revisão das caches e sua operação. Porém, este capítulo descreve inovações mais avançadas, que focam a diferença de desempenho processador-memória.

Quando uma palavra não é encontrada na cache, ela precisa ser recuperada de um nível inferior na hierarquia (que pode ser outra cache ou a memória principal) e colocada na

cache antes de continuar. Múltiplas palavras, chamadas *bloco* (ou *linha*), são movidas por questões de eficiência, e porque elas provavelmente serão necessárias em breve, devido à localização espacial. Cada bloco da I-cache inclui uma *tag* para ver a qual endereço de memória ela corresponde.

Uma decisão de projeto importante é em que parte da cache os blocos (ou linhas) podem ser colocados. O esquema mais popular é a *associação por conjunto* (*set associative*), em que um *conjunto* é um grupo de blocos na cache. Primeiro, um bloco é mapeado para um conjunto; depois pode ser colocado em qualquer lugar dentro desse conjunto. Encontrar um bloco consiste primeiramente em mapear o endereço do bloco para o conjunto e depois em examinar o conjunto — normalmente em paralelo — para descobrir o bloco. O conjunto é escolhido pelo endereço dos dados:

$$(\text{Endereço do bloco})\ \text{MOD}\ (\text{Número de conjuntos da cache})$$

Se houver *n* blocos em um conjunto, o posicionamento da cache será denominado *associativo por conjunto com n vias* (*n-way set associative*). As extremidades da associatividade do conjunto têm seus próprios nomes. Uma cache *mapeada diretamente* tem apenas um bloco por conjunto (de modo que um bloco sempre é colocado no mesmo local), e uma cache *totalmente associativa* tem apenas um conjunto (de modo que um bloco pode ser colocado em qualquer lugar).

O caching de dados que são apenas lidos é fácil, pois as cópias na cache e na memória são idênticas. O caching de escritas é mais difícil: como a cópia na cache e na memória pode ser mantida consistente? Existem duas estratégias principais. A *write-through*, que atualiza o item na cache *e* também escreve na memória principal, para atualizá-la. A *write-back* só atualiza a cópia na cache. Quando o bloco está para ser atualizado, ele é copiado de volta na memória. As duas estratégias de escrita podem usar um *buffer de escrita* para permitir que a cache prossiga assim que os dados forem colocados no buffer, em vez de esperar a latência total para escrever os dados na memória.

Uma medida dos benefícios de diferentes organizações de cache é a taxa de falta. A *taxa de falta* (*miss rate*) é simplesmente a fração de acessos à cache que resulta em uma falta, ou seja, o número de acessos em que ocorre a falta dividido pelo número total de acessos.

Para entender as causas das altas taxas de falta, que podem inspirar projetos de cache melhores, o modelo dos três C classifica todas as faltas em três categorias simples:

- *Compulsória.* O primeiro acesso a um bloco *não pode* estar na cache, de modo que o bloco precisa ser trazido para a cache. As faltas compulsórias são aquelas que ocorrem mesmo que se tenha uma I-cache infinita.
- *Capacidade.* Se a cache tiver todos os blocos necessários durante a execução de≈um programa, as faltas por capacidade (além das faltas compulsórias) ocorrerão porque os blocos são descartados e mais tarde recuperados.
- *Conflito.* Se a estrutura de colocação do bloco não for totalmente associativa, faltas por conflito (além das faltas compulsórias e de capacidade) ocorrerão porque um bloco pode ser descartado e mais tarde recuperado se os blocos em conflito forem mapeados para o seu conjunto e os acessos aos diferentes blocos forem intercalados.

As Figuras B.8 e B.9, nas páginas B-21 e B-22, mostram a frequência relativa das faltas de cache desmembradas pelos "três C". Como veremos nos Capítulos 3 e 5, o multithreading e os múltiplos núcleos acrescentam complicações para as caches, tanto aumentando o potencial para as faltas de capacidade quanto acrescentando um quarto C para as faltas de *coerência* advindas de esvaziamentos de cache, a fim de manter múltiplas caches coerentes em um multiprocessador. Vamos considerar esses problemas no Capítulo 5.

Infelizmente, a taxa de falta pode ser uma medida confusa por vários motivos. Logo, alguns projetistas preferem medir as *faltas por instrução* em vez das faltas por referência de memória (taxa de falta). Essas duas estão relacionadas:

$$\frac{\text{Perdas}}{\text{Instrução}} = \frac{\text{Taxas de perdas} \times \text{Acessos à memória}}{\text{Contagem de instruções}} = \text{Taxa de perda} \times \frac{\text{Acessos à memória}}{\text{Instrução}}$$

(Normalmente são relatadas como faltas por 1.000 instruções, para usar inteiros no lugar de frações.)

O problema com as duas medidas é que elas não levam em conta o custo de uma falta. Uma medida melhor é o *tempo de acesso médio à memór*ia:

$$\text{Tempo de acesso médio à memória} = \text{Tempo de acerto} + \text{Taxa de falta} \times \text{Penalidade de falta}$$

onde *tempo de acerto* é o tempo de acesso quando o item acessado está na cache e *penalidade de falta* é o tempo para substituir o bloco de memória (ou seja, o custo de uma falta). O tempo de acesso médio à memória ainda é uma medida indireta do desempenho; embora sendo uma medida melhor do que a taxa de falta, ainda não é um substituto para o tempo de execução. No Capítulo 3, veremos que os processadores especulativos podem executar outras instruções durante uma falta, reduzindo assim a penalidade efetiva de falta. O uso de multithreading (apresentado no Cap. 3) também permite que um processador tolere faltas sem ser forçado a ficar inativo. Como veremos em breve, para tirar vantagem de tais técnicas de tolerância de latência, precisamos de caches que possam atender requisições e, ao mesmo tempo, lidar com uma falta proeminente.

Se este material é novo para você ou se esta revisão estiver avançando muito rapidamente, consulte o Apêndice B. Ele aborda o mesmo material introdutório com profundidade e inclui exemplos de caches de computadores reais e avaliações quantitativas de sua eficácia.

A Seção B.3, no Apêndice B, também apresenta seis otimizações de cache básicas, que revisamos rapidamente aqui. O apêndice oferece exemplos quantitativos dos benefícios dessas otimizações.

1. *Tamanho de bloco maior para reduzir a taxa de falta.* O modo mais simples de reduzir a taxa de falta é tirar proveito da proximidade espacial e aumentar o tamanho do bloco. Blocos maiores reduzem as faltas compulsórias, mas também aumentam a penalidade da falta. Já que blocos maiores diminuem o número de tags, eles podem reduzir ligeiramente a potência estática. Blocos de tamanhos maiores também podem aumentar as faltas por capacidade ou conflito, especialmente em caches menores. Selecionar o tamanho de bloco correto é uma escolha complexa que depende do tamanho da cache e da penalidade de falta.
2. *Caches maiores para reduzir a taxa de falta.* O modo óbvio de reduzir as faltas por capacidade é aumentar a capacidade da cache. As desvantagens incluem o tempo de acerto potencialmente maior da memória de cache maior, além de custo e consumo de potência mais altos. Caches maiores aumentam tanto a potência dinâmica quanto a estática.
3. *Associatividade mais alta para reduzir a taxa de falta.* Obviamente, aumentar a associatividade reduz as faltas por conflito. Uma associatividade maior pode ter o custo de maior tempo de acerto. Como veremos em breve, a associatividade também aumenta o potência.
4. *Caches multiníveis para reduzir a penalidade de falta.* Uma decisão difícil é a de tornar o tempo de acerto da cache rápido, para acompanhar a taxa de clock crescente dos processadores ou tornar a cache grande, para contornar a grande diferença entre o

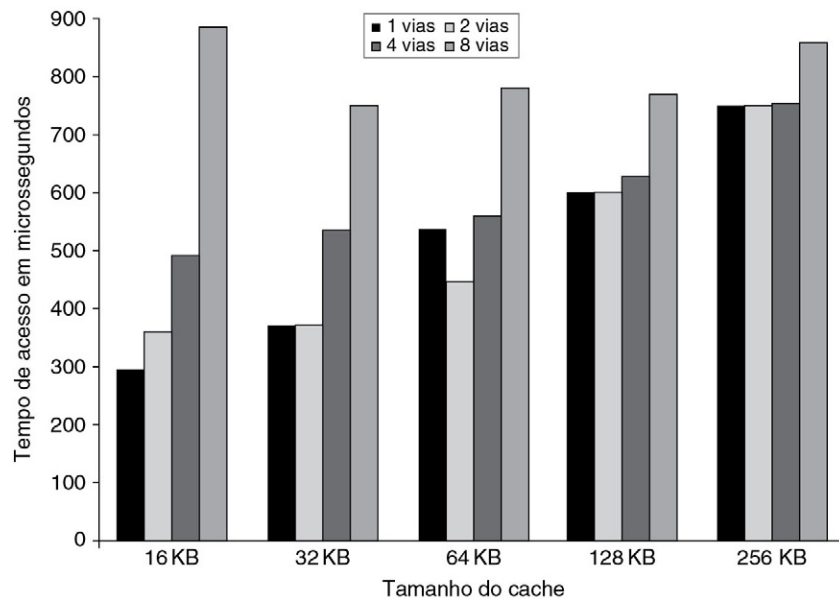



FIGURA 2.3 **Os tempos de acesso geralmente aumentam conforme o tamanho da cache e a associatividade aumentam.**

Esses dados vêm do CACTI modelo 6.5 de Tarjan, Thoziyoor e Jouppi (2005). O dado supõe um tamanho característico de 40 nm (que está entre a tecnologia usada nas versões mais rápida e segunda mais rápida do Intel i7 e igual à tecnologia usada nos processadores AMD embutidos mais velozes), um único banco e blocos de 64 bytes. As suposições sobre o leiaute da cache e as escolhas complexas entre atrasos de interconexão (que dependem do tamanho do bloco de cache sendo acessado) e o custo de verificações de tag e multiplexação levaram a resultados que são ocasionalmente surpreendentes, como o menor tempo de acesso de uma associatividade por conjunto de duas vias com 64 KB em comparação com o mapeamento direto. De modo similar, os resultados com associatividade por conjunto de oito vias gera um comportamento incomum conforme o tamanho da cache aumenta. Uma vez que tais observações são muito dependentes da tecnologia e suposições detalhadas de projeto, ferramentas como o CACTI servem para reduzir o espaço de busca, e não para uma análise precisa das opções.

processador e a memória principal. A inclusão de outro nível de cache entre a cache original e a memória simplifica a decisão (Fig. 2.3). A cache de primeiro nível pode ser pequena o suficiente para combinar com um tempo de ciclo de clock rápido, enquanto a cache de segundo nível pode ser grande o suficiente para capturar muitos acessos que iriam para a memória principal. O foco nas faltas nas caches de segundo nível leva a blocos maiores, capacidade maior e associatividade mais alta. Se L1 e L2 se referem, respectivamente, às caches de primeiro e segundo níveis, podemos redefinir o tempo de acesso médio à memória:

$$\text{Tempo acerto}_{L1} + \text{Taxa falta}_{L1} \times (\text{Tempo acerto}_{L2} + \text{Taxa falta}_{L2} \times \text{Penalidade falta}_{L2})$$

5. Dar pri*oridade às faltas de leitura, em vez de escrita, para reduzir a penalidade de falta.* Um buffer de escrita é um bom lugar para implementar essa otimização. Os buffers de escrita criam riscos porque mantêm o valor atualizado de um local necessário em uma falta de leitura, ou seja, um risco de leitura após escrita pela memória. Uma solução é verificar o conteúdo do buffer de escrita em uma falta de leitura. Se não houver conflitos e se o sistema de memória estiver disponível, o envio da leitura antes das escritas reduzirá a penalidade de falta. A maioria dos processadores dá prioridade às leituras em vez de às escritas. Essa escolha tem pouco efeito sobre o consumo de potência.

6. *Evitar tradução de endereço durante a indexação da cache para reduzir o tempo de acerto.* As caches precisam lidar com a tradução de um endereço virtual do processador para um endereço físico para acessar a memória (a memória virtual é explicada nas Seções 2.4 e B.4). Uma otimização comum é usar o offset de página — a parte idêntica nos endereços virtual e físico — para indexar a cache, como descrito no Apêndice B, página B-34. Esse método de índice virtual/tag físico introduz algumas complicações de sistema e/ou limitações no tamanho e estrutura da cache L1, mas as vantagens de remover o acesso ao *translation buffer lookaside* (TLB) do caminho crítico supera as desvantagens.

Observe que cada uma dessas seis otimizações possui uma desvantagem em potencial, que pode levar a um tempo de acesso médio à memória ainda maior em vez de diminuí-lo.

O restante deste capítulo considera uma familiaridade com o material anterior e os detalhes apresentados no Apêndice B. Na seção "Juntando tudo", examinamos a hierarquia de memória para um microprocessador projetado para um servidor de alto nível, o Intel Core i7, além de um projetado para uso em um PMD, o Arm Cortex-A8, que é a base para o processador usado no Apple iPad e diversos smartphones de alto nível. Dentro de cada uma dessas classes existe significativa diversidade na abordagem, devido ao uso planejado do computador. Embora o processador de alto nível usado no servidor tenha mais núcleos e caches maiores do que os processadores Intel projetados para usos em desktop, os processadores têm arquiteturas similares. As diferenças são guiadas pelo desempenho e pela natureza da carga de trabalho. Computadores desktop executam primordialmente um aplicativo por vez sobre um sistema operacional para um único usuário, enquanto computadores servidores podem ter centenas de usuários rodando dúzias de aplicações ao mesmo tempo. Devido a essas diferenças na carga de trabalho, os computadores desktop geralmente se preocupam mais com a latência média da hierarquia de memória, enquanto os servidores se preocupam também com a largura de banda da memória. Mesmo dentro da classe de computadores desktop, existe grande diversidade entre os netbooks, que vão desde os de baixo nível com processadores reduzidos mais similares aos encontrados em PMDs de alto nível até desktops de alto nível cujos processadores contêm múltiplos núcleos e cuja organização lembra a de um servidor de baixo nível.

Em contraste, os PMDs não só atendem a um usuário, mas geralmente também têm sistemas operacionais menores, geralmente menos multitasking (a execução simultânea de diversas aplicações) e aplicações mais simples. Em geral, os PMDs também usam memória Flash no lugar de discos, e a maioria considera tanto o desempenho quanto o consumo de energia, que determina a vida da bateria.

2.2 DEZ OTIMIZAÇÕES AVANÇADAS DE DESEMPENHO DA CACHE

A fórmula do tempo médio de acesso à memória, dada anteriormente, nos oferece três medidas para otimizações da cache: tempo de acerto, taxa de falta e penalidade de falta. Dadas as tendências atuais, adicionamos largura de banda da cache e consumo de potência a essa lista. Podemos classificar as 10 otimizações avançadas de cache que vamos examinar em cinco categorias baseadas nessas medidas:

1. *Reduzir o tempo de acerto*: caches de primeiro nível pequenas e simples e previsão de vias (way-prediction). Ambas as técnicas diminuem o consumo de potência.
2. *Aumentar a largura de banda da cache*: caches em pipeline, caches em multibanco e caches de não bloqueio. Essas técnicas têm impacto variado sobre o consumo de potência.

3. *Reduzir a penalidade de falta*: primeira palavra crítica e utilização de write buffer merges. Essas otimizações têm pouco impacto sobre a potência.
4. *Reduzir a taxa de falta*: otimizações do compilador. Obviamente, qualquer melhoria no tempo de compilação melhora o consumo de potência.
5. *Reduzir a penalidade de falta ou a taxa de falta por meio do paralelismo*: pré-busca do hardware e pré-busca do compilador. Essas otimizações geralmente aumentam o consumo de potência, principalmente devido aos dados pré-obtidos que não são usados.

Em geral, a complexidade do hardware aumenta conforme prosseguimos por essas otimizações. Além disso, várias delas requerem uma tecnologia complexa de compiladores. Concluíremos com um resumo da complexidade da implementação e os benefícios das 10 técnicas (Fig. 2.11, na página 82) para o desempenho. Uma vez que algumas delas são bastantes diretas, vamos abordá-las rapidamente. Outras, contudo, requerem descrições mais detalhadas.

Primeira otimização: caches pequenas e simples para reduzir o tempo de acerto e a potência

A pressão de um ciclo de clock rápido e das limitações de consumo de potência encoraja o tamanho limitado das caches de primeiro nível. Do mesmo modo, o uso de níveis menores de associatividade pode reduzir tanto o tempo de acerto quanto a potência, embora tais relações sejam mais complexas do que aquelas envolvendo o tamanho.

O caminho crítico de tempo em um acerto de cache é um processo, em três etapas, de endereçar a memória de tag usando a parte do índice do endereço, comparar o valor de tag de leitura ao endereço e configurar o multiplexador para selecionar o item de dados correto se a cache for configurada como associativa. Caches mapeadas diretamente podem sobrepor a verificação de tag à transmissão dos dados, reduzindo efetivamente o tempo de acerto. Além do mais, níveis inferiores de associatividade geralmente vão reduzir o consumo de potência, porque menos linhas de cache devem ser acessadas.

Embora a quantidade de cache no chip tenha aumentado drasticamente com as novas gerações de microprocessadores, graças ao impacto da taxa de clock devido a uma cache L1 maior, recentemente o tamanho das caches aumentou muito pouco ou nada. Em muitos processadores recentes, os projetistas optaram por maior associatividade em vez de caches maiores. Uma consideração adicional na escolha da associatividade é a possibilidade de eliminar as instâncias de endereços (*address aliases*). Vamos discutir isto em breve.

Uma técnica para determinar o impacto sobre o tempo de acerto e a potência antes da montagem de um chip é a utilização de ferramentas CAD. O CACTI é um programa para estimar o tempo de acesso e a potência de estruturas de cache alternativas nos microprocessadores CMOS, dentro de 10% das ferramentas de CAD mais detalhadas. Para determinada característica de tamanho mínimo, o CACTI estima o tempo de acerto das caches quando variam o tamanho da cache, a associatividade e o número de portas de leitura/escrita e parâmetros mais complexos. A Figura 2.3 mostra o impacto estimado sobre o tempo de acerto quando o tamanho da cache e a associatividade são variados. Dependendo do tamanho da cache, para esses parâmetros o modelo sugere que o tempo de acerto para o mapeamento direto é ligeiramente mais rápido do que a associatividade por conjunto com duas vias, que a associatividade por conjunto com duas vias é 1,2 vez mais rápida do que com quatro vias, e com quatro vias é 1,4 vez mais rápido do que a associatividade com oito vias. Obviamente, essas estimativas dependem da tecnologia e do tamanho da cache.

Exemplo Usando os dados da Figura B.8, no Apêndice B, e da Figura 2.3, determine se uma cache L1 de 32 KB, associativa por conjunto com quatro vias tem tempo de acesso à memória mais rápido do que uma cache L1 de 32 KB, associativa por conjunto com quatro vias. Suponha que a penalidade de falta para a cache L2 seja 15 vezes o tempo de acesso para a cache L1 mais rápido. Ignore as faltas além de L2. Qual é o tempo médio de acesso à memória mais rápido?

Resposta Seja o tempo de acesso para cache com associatividade por conjunto de duas vias igual a 1. Então, para a cache de duas vias:

$$\text{Tempo de acesso médio à memória}_{2\text{vias}} = \text{Tempo de acerto} + \text{Tempo de falta} \times \text{Penalidade de falta}$$
$$= 1 + 0{,}038 \times 15 = 1{,}38$$

Para a cache de quatro vias, o tempo de clock é 1,4 vez maior. O tempo gasto da penalidade de falta é 15/1,4 = 10,1. Por simplicidade, assuma que ele é igual a 10:

$$\text{Tempo de acesso médio à memória}_{4\text{vias}} = \text{Tempo de acerto}_{2\text{vias}} \times 1{,}4 + \text{Taxa de perda} \times \text{Penalidade de falta}$$
$$= 1{,}4 + 0{,}037 \times 10 = 1{,}77$$

Obviamente, a maior associatividade parece uma troca ruim. Entretanto, uma vez que o acesso à cache nos processadores modernos muitas vezes é pipelined, o impacto exato sobre o tempo do ciclo de clock é difícil de avaliar.

O consumo de energia também deve ser considerado na escolha tanto do tamanho da cache como na da associatividade, como mostra a Figura 2.4. O custo energético da maior associatividade varia desde um fator de mais de 2 até valores irrelevantes, em caches de 128 KB ou 256 KB, indo do mapeado diretamente para associatividade por conjunto de duas vias.

Em projetos recentes, três fatores levaram ao uso de maior associatividade nas caches de primeiro nível: 1) muitos processadores levam pelo menos dois ciclos de clock para acessar a cache, por isso o impacto de um período de tempo mais longo pode não ser

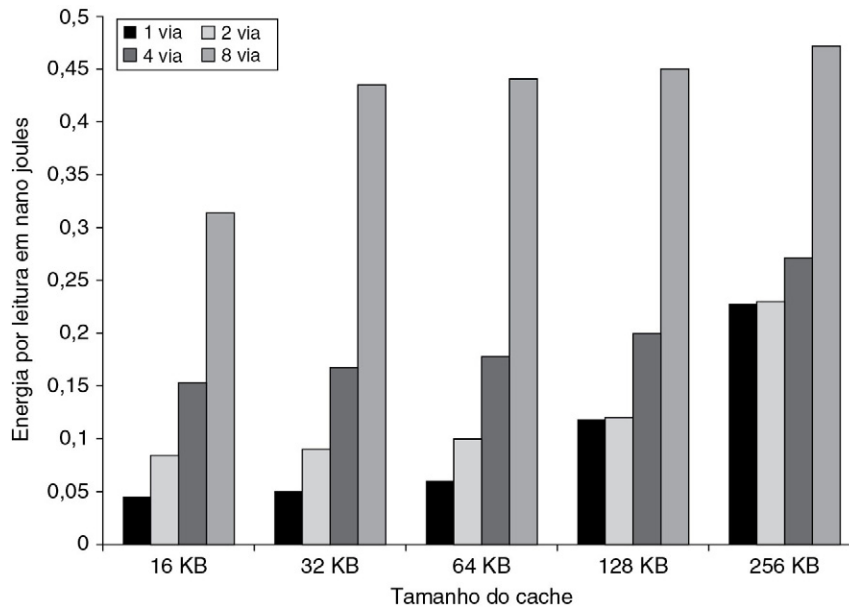



FIGURA 2.4 **O consumo de energia por leitura aumenta conforme aumentam o tamanho da cache e a associatividade.**

Como na figura anterior, o CACTI é usado para o modelamento com os mesmos parâmetros tecnológicos. A grande penalidade para as caches associativas por conjunto de oito vias é decorrente do custo de leitura de oito tags e aos dados correspondentes em paralelo.

crítico; 2) para manter o TLB fora do caminho crítico (um atraso que seria maior do que aquele associado à maior associatividade), embora todas as caches L1 devam ser indexadas virtualmente. Isso limita o tamanho da cache para o tamanho da página vezes a associatividade, porque somente os bits dentro da página são usados para o índice. Existem outras soluções para o problema da indexação da cache antes que a tradução do endereço seja completada, mas aumentar a associatividade, que também tem outros benefícios, é a mais atraente; 3) com a introdução do multithreading (Cap. 3), as faltas de conflito podem aumentar, tornando a maior associatividade mais atraente.

Segunda otimização: previsão de via para reduzir o tempo de acesso

Outra técnica reduz as faltas por conflito e ainda mantém a velocidade de acerto da cache mapeada diretamente. Na *previsão de via*, bits extras são mantidos na cache para prever a via ou o bloco dentro do conjunto do *próximo* acesso à cache. Essa previsão significa que o multiplexador é acionado mais cedo para selecionar o bloco desejado, e apenas uma comparação de tag é realizada nesse ciclo de clock, em paralelo com a leitura dos dados da cache. Uma falta resulta na verificação de outros blocos em busca de combinações no próximo ciclo de clock.

Acrescentados a cada bloco de uma cache estão os bits de previsão de bloco. Os bits selecionam quais dos blocos experimentar no *próximo* acesso à cache. Se a previsão for correta, a latência de acesso à cache será o tempo de acerto rápido. Se não, ele tentará o outro bloco, mudará o previsor de via e terá uma latência extra de um ciclo de clock. As simulações sugeriram que a exatidão da previsão de conjunto excede 90% para um conjunto de duas vias e em 80% para um conjunto de quatro vias, com melhor precisão em caches de instruções (I-caches) do que em caches de dados (D-caches). A previsão de via gera menos tempo médio de acesso à memória para um conjunto de duas vias se ele for pelo menos 10% mais rápido, o que é muito provável. A previsão de via foi usada pela primeira vez no MIPS R10000 em meados dos anos 1990. Ela é muito popular em processadores que empregam associatividade por conjunto de duas vias e é usada no ARM Cortex-A8 com caches associativas por conjunto de quatro vias. Para processadores muito rápidos, pode ser desafiador implementar o atraso de um ciclo que é essencial para manter uma penalidade pequena de previsão de via.

Uma forma estendida de previsão de via também pode ser usada para reduzir o consumo de energia usando os bits de previsão de via para decidir que bloco de cache acessar na verdade (os bits de previsão de via são essencialmente bits de endereço adicionais). Essa abordagem, que pode ser chamada *seleção de via*, economiza energia quando a previsão de via está correta, mas adiciona um tempo significativo a uma previsão incorreta de via, já que o acesso, e não só a comparação e a seleção de tag, deve ser repetida. Tal otimização provavelmente faz sentido somente em processadores de baixa potência. Inoue, Ishihara e Muramaki (1999) estimaram que o uso da técnica da seleção de via em uma cache associativas por conjunto aumenta o tempo médio de acesso para a I-cache em 1,04 e em 1,13 para a D-cache, nos benchmarks SPEC95, mas gera um consumo médio de energia de cache de 0,28 para a I-cache e 0,35 para a D-cache. Uma desvantagem significativa da seleção de via é que ela torna difícil o pipeline do acesso à cache.

Exemplo Suponha que os acessos à D-cache sejam a metade dos acessos à I-cache, e que a I-cache e a D-cache sejam responsáveis por 25% e 15% do consumo de energia do processador em uma implementação normal associativa por conjunto de quatro vias. Determine que seleção de via melhora o desempenho por watt com base nas estimativas do estudo anterior.

Resposta Para a I-cache, a economia é de 25 × 0,28 = 0,07 potência total, enquanto para a D-cache é de 15 × 0,35 = 0,05 para uma economia total de 0,12. A versão com previsão de via requer 0,88 do requisito de potência da cache padrão de quatro vias. O aumento no tempo de acesso à cache é o aumento no tempo médio de acesso à I-cache, mais metade do aumento do tempo de acesso à D-cache, ou 1,04 + 0,5 × 0,13 = 1,11 vezes mais demorado. Esse resultado significa que a seleção de via tem 0,090 do desempenho de uma cache padrão de quatro vias. Assim, a seleção de via melhora muito ligeiramente o desempenho por joule por uma razão de 0,90/0,88 = 1,02. Essa otimização é mais bem usada onde a potência, e não o desempenho, é o objetivo principal.

Terceira otimização: acesso à cache em pipeline para aumentar a largura de banda da cache

Essa otimização serve simplesmente para possibilitar o acesso pipeline à cache, de modo que a latência efetiva de um acerto em uma cache de primeiro nível possa ser de múltiplos ciclos de clock, gerando rápidos ciclos de clock e grande largura de banda, mas com acertos lentos. Por exemplo, em meados dos anos 1990, o pipeline para o processador Intel Pentium usava um ciclo de clock para acessar a cache de instruções; de meados dos anos 1990 até o ano 2000, levava dois ciclos para o Pentium Pro ao Pentium III; para o Pentium 4, lançado em 2000, e o Intel Core i7 atual leva quatro ciclos de clock. Essa mudança aumenta o número de estágios da pipeline, levando a uma penalidade maior nos desvios mal previstos e mais ciclos de clock entre o carregamento e o uso dos dados (Cap. 3), mas isso torna mais fácil incorporar altos graus de associatividade.

Quarta otimização: caches sem bloqueio para aumentar a largura de banda da cache

Para os computadores com pipeline que permitem a execução fora de ordem (discutidos no Cap. 3), o processador não precisa parar (stall) em uma falta na cache de dados, esperando o dado. Por exemplo, o processador pode continuar buscando instruções da cache de instruções enquanto espera que a cache de dados retorne os dados que faltam. Uma cache *sem bloqueio* ou *cache sem travamento* aumenta os benefícios em potencial de tal esquema, permitindo que a cache de dados continue a fornecer acertos de cache durante uma falta. Essa otimização de "acerto sob falta" reduz a penalidade de falta efetiva, sendo útil durante uma falta, em vez de ignorar as solicitações do processador. Uma opção sutil e complexa é que a cache pode reduzir ainda mais a penalidade de falta efetiva se puder sobrepor múltiplas faltas: uma otimização "acerto sob múltiplas faltas" ou "falta sob falta". A segunda opção só será benéfica se o sistema de memória puder atender a múltiplas faltas. A maior parte dos processadores de alto desempenho (como o Intel Core i7) geralmente suporta ambos, enquanto os processadores de baixo nível, como o ARM A8, fornecem somente suporte limitado sem bloqueio no L2.

Para examinar a eficiência das caches sem bloqueio na redução da penalidade de falta de caches, Farkas e Jouppi (1994) realizaram um estudo assumindo caches de 8 KB com uma penalidade de falta de 14 ciclos. Eles observaram uma redução na penalidade efetiva de falta de 20% para os benchmarks SPECINT92 e de 30% para os benchmarks SPECFP92 ao permitir um acerto sob falta.

Recentemente, Li, Chen, Brockman e Jouppi (2011) atualizaram esse estudo para usar uma cache multiníveis, suposições mais modernas sobre as penalidades de falta, além dos benchmarks SPEC2006, maiores e mais exigentes. O estudo foi feito supondo um modelo baseado em um único núcleo de um Intel i7 (Seção 2.6) executando os benchmarks SPC2006. A Figura 2.5 mostra a redução na latência de acesso à cache de dados quando

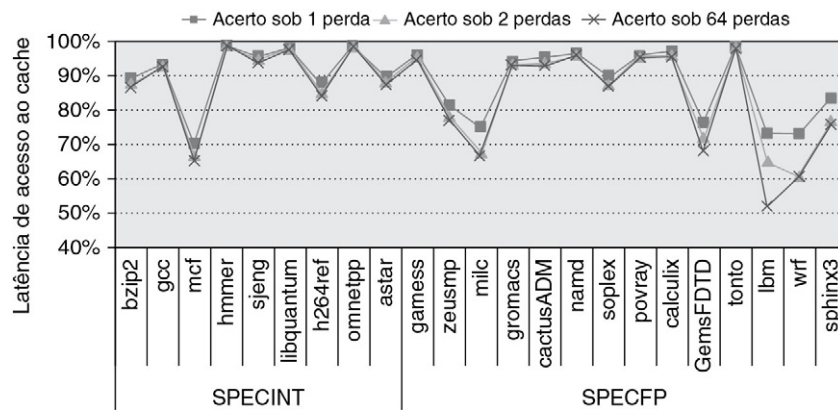



FIGURA 2.5 **A eficácia de uma cache sem bloqueio é avaliada, permitindo 1, 2 ou 64 acertos sob uma falta de cache com os benchmarks 9 SPCINT (à esquerda) e 9 SECFP (à direita).**

O sistema de memória de dados modelado com base no Intel i7 consiste em uma cache L1 de 32 KB, com latência de acesso de quatro ciclos. A cache L2 (compartilhada com instruções) é de 256 KB, com uma latência de acesso de 10 ciclos de clock. O L3 tem 2 MB e uma latência de acesso de 32 ciclos. Todos as caches são associativas por conjunto de oito vias e têm tamanho de bloco de 64 bytes. Permitir um acerto sob falta reduz a penalidade de falta em 9% para os benchmarks inteiros e de 12,5% para os de ponto flutuante. Permitir um segundo acerto melhora esses resultados para 10% e 16%, e permitir 64 resulta em pouca melhoria adicional.

permitimos 1, 2 e 64 acertos sob uma falta. A legenda apresenta mais detalhes sobre o sistema de memória. As caches maiores e a adição de uma cache L3 desde o estudo anterior reduziram os benefícios com os benchmarks SPECINT2006, mostrando uma redução média na latência da cache de cerca de 9%, e com os benchmarks SPECFP2006, de cerca de 12,5%.

Exemplo O que é mais importante para os programas de ponto flutuante: associatividade por conjunto em duas vias ou acerto sob uma falta? E para os programas de inteiros? Considere as taxas médias de falta a seguir para caches de dados de 32 KB: 5,2% para programas de ponto flutuante com cache de mapeamento direto, 4,9% para esses programas com cache com associatividade por conjunto em duas vias, 3,5% para programas de inteiros com cache mapeado diretamente e 3,2% para programas de inteiros com cache com associatividade por conjunto em duas vias. Considere que a penalidade de falta para L2 é de 10 ciclos e que as faltas e acertos do L2 sejam os mesmos.

Resposta Para programas de ponto flutuante, os tempos de stall médios da memória são:

$$\text{Taxa de falta}_{DM} \times \text{Penalidade de falta} = 5{,}2\% \times 10 = 0{,}52$$
$$\text{Taxa de falta}_{2\text{ vias}} \times \text{Penalidade de falta} = 4{,}9\% \times 10 = 0{,}49$$

A latência de acesso à cache (incluindo as paradas — stalls) para associatividade de duas é 0,49/0,52 ou 94% da cache mapeada diretamente. A legenda da Figura 2.5 revela que o acerto sob uma falta reduz o tempo médio de stall da memória para 87,5% de uma cache com bloqueio. Logo, para programas de ponto flutuante, a cache de dados com mapeamento direto com suporte para acerto sob uma falta oferece melhor desempenho do que uma cache com associatividade por conjunto em duas vias, que bloqueia em uma falta.

Para programas inteiros, o cálculo é:

$$\text{Taxa de falta}_{DM} \times \text{Penalidade de falta} = 3{,}5\% \times 10 = 0{,}35$$
$$\text{Taxa de falta}_{2\text{ vias}} \times \text{Penalidade de falta} = 3{,}2\% \times 10 = 0{,}32$$

A latência de acesso à cache com associatividade por conjunto de duas vias é, assim, 0,32/0,35 ou 91% de cache com mapeamento direto, enquanto a redução na latência de acesso, quando permitimos um acerto sob falta, é de 9%, tornando as duas opções aproximadamente iguais.

A dificuldade real com a avaliação do desempenho das caches de não bloqueio é que uma falta de cache não causa necessariamente um stall no processador. Nesse caso, é difícil julgar o impacto de qualquer falta isolada e, portanto, é difícil calcular o tempo de acesso médio à memória. A penalidade de falta efetiva não é a soma das faltas, mas o tempo não sobreposto em que o processador é adiado. O benefício das caches de não bloqueio é complexo, pois depende da penalidade de falta quando ocorrem múltiplas faltas, do padrão de referência da memória e de quantas instruções o processador puder executar com uma falta pendente.

Em geral, os processadores fora de ordem são capazes de ocultar grande parte da penalidade de falta de uma falta de cache de dados L1 na cache L2, mas não são capazes de ocultar uma fração significativa de uma falta de cache de nível inferior. Decidir quantas faltas pendentes suportar depende de diversos fatores:

- A localidade temporal e espacial no fluxo da falta, que determina se uma falta pode iniciar um novo acesso a um nível inferior da cache ou à memória.
- A largura da banda da resposta da memória ou da cache.
- Permitir mais faltas pendentes no nível mais baixo da cache (onde o tempo de falta é o mais longo) requer pelo menos o suporte do mesmo número de faltas em um nível mais alto, já que a falta deve se iniciar na cache de nível mais alto.
- A latência do sistema de memória.

O exemplo simplificado a seguir mostra a ideia principal.

Exemplo Considere um tempo de acesso à memória principal de 36 ns e um sistema de memória capaz de uma taxa sustentável de transferência de 16 GB/s. Se o tamanho do bloco for de 64 bytes, qual será o número máximo de faltas pendentes que precisamos suportar, supondo que possamos manter o pico de largura de banda, dado o fluxo de requisições, e que os acessos nunca entram em conflito? Se a probabilidade de uma referência colidir com uma das quatro anteriores for de 50% e supondo-se que o acesso tenha que esperar até que o acesso anterior seja completado, estime o número máximo de referências pendentes. Para simplificar, ignore o tempo entre as faltas.

Resposta No primeiro caso, supondo que possamos manter o pico de largura de banda, o sistema de memória pode suportar $(16 \times 10)^9/64 = 250$ milhões de referências por segundo. Uma vez que cada referência leva 36 ns, podemos suportar $250 \times 10^6 \times 36 \times 10^{-9}$ = nove referências. Se a probabilidade de uma colisão for maior do que 0, então precisamos de mais referências pendentes, uma vez que não podemos começar a trabalhar nessas referências. O sistema de memória precisa de mais referências independentes, não de menos! Para aproximar isso, podemos simplesmente supor que é preciso que a metade das referências de memória não seja enviada para a memória. Isso quer dizer que devemos suportar duas vezes mais referências pendentes, ou 18.

Em seu estudo, Li, Chen, Brosckman e Jouppi descobriram que a redução na CPI para os programas de inteiros foi de cerca de 7% para um acerto sob falta e cerca de 12,7% para 64. Para os programas de ponto flutuante, as reduções foram de 12,7% para um acerto sob falta e de 17,8% para 64. Essas reduções acompanham razoavelmente de perto as reduções na latência no acesso à cache de dados mostrado na Figura 2.5.

Quinta otimização: caches multibanco para aumentar a largura de banda da cache

Em vez de tratar a cache como um único bloco monolítico, podemos dividi-lo em bancos independentes que possam dar suporte a acessos simultâneos. Os bancos foram usados

Endereço do bloco	Banco 0	Endereço do bloco	Banco 1	Endereço do bloco	Banco 2	Endereço do bloco	Banco 3
0		1		2		3	
4		5		6		7	
8		9		10		11	
12		13		14		15	

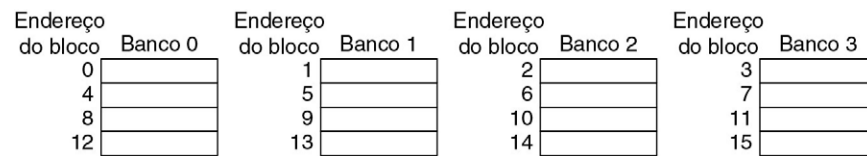



FIGURA 2.6 Bancos de caches intercalados em quatro vias, usando o endereçamento de bloco.
Considerando 64 bytes por bloco, cada um desses endereços seria multiplicado por 64 para obter o endereçamento do byte.

originalmente para melhorar o desempenho da memória principal e agora são usados tanto nos modernos chips de DRAM como nas caches. O Arm Cortex-A8 suporta 1-4 bancos em sua cache L2; o Intel Core i7 tem quatro bancos no L1 (para suportar até dois acessos de memória por clock), e o L2 tem oito bancos.

Obviamente, o uso de bancos funciona melhor quando os acessos se espalham naturalmente por eles, de modo que o mapeamento de endereços a bancos afeta o comportamento do sistema de memória. Um mapeamento simples, que funciona bem, consiste em espalhar os endereços do bloco sequencialmente pelos bancos, algo chamado *intercalação sequencial*. Por exemplo, se houver quatro bancos, o banco 0 terá todos os blocos cujo endereço módulo 4 é igual a 0; o banco 1 terá todos os blocos cujo endereço módulo 4 é 1, e assim por diante. A Figura 2.6 mostra essa intercalação. Bancos múltiplos também são um modo de reduzir o consumo de energia, tanto nas caches quanto na DRAM.

Sexta otimização: palavra crítica primeiro e reinício antecipado para reduzir a penalidade da falta

Essa técnica é baseada na observação de que o processador normalmente precisa de apenas uma palavra do bloco de cada vez. Essa estratégia é a da impaciência: não espere até que o bloco inteiro seja carregado para então enviar a palavra solicitada e reiniciar o processador. Aqui estão duas estratégias específicas:

- *Palavra crítica primeiro.* Solicite primeiro a palavra que falta e envie-a para o processador assim que ela chegar; deixe o processador continuar a execução enquanto preenche o restante das palavras no bloco.
- *Reinício antecipado.* Busque as palavras na ordem normal, mas, assim que a palavra solicitada chegar, envie-a para o processador e deixe que ele continue a execução.

Geralmente, essas técnicas só beneficiam projetos com grandes blocos de cache, pois o benefício é baixo, a menos que os blocos sejam grandes. Observe que, normalmente, as caches continuam a satisfazer os acessos a outros blocos enquanto o restante do bloco está sendo preenchido.

Infelizmente, dada a proximidade espacial, existe boa chance de que a próxima referência sirva para o restante do bloco. Assim como as caches de não bloqueio, a penalidade de falta não é simples de calcular. Quando existe uma segunda solicitação da palavra crítica primeiro, a penalidade de falta efetiva é o tempo não sobreposto da referência até que a segunda parte chegue. Os benefícios da palavra crítica primeiro e de reinício antecipado dependem do tamanho do bloco e da probabilidade de outro acesso à parte do bloco que ainda não foi acessada.

Sétima otimização: write buffer merge de escrita para reduzir a penalidade de falta

Caches write-through contam com buffers de escrita, pois todos os armazenamentos precisam ser enviados para o próximo nível inferior da hierarquia. Até mesmo as caches

write-back utilizam um buffer simples quando um bloco é substituído. Se o buffer de escrita estiver vazio, os dados e o endereço completo serão escritos no buffer, e a escrita será terminada do ponto de vista do processador; o processador continuará trabalhando enquanto o buffer de escrita se prepara para escrever a palavra na memória. Se o buffer tiver outros blocos modificados, os endereços poderão ser verificados para saber se o endereço desses novos dados combina com o endereço de uma entrada válida no buffer de escrita. Então, os novos dados serão combinados com essa entrada. A *mesclagem de escrita* é o nome dessa otimização. O Intel Core i7, entre muitos outros, utiliza a mesclagem de escrita.

Se o buffer estiver cheio e não houver combinação de endereço, a cache (e o processador) precisará esperar até que o buffer tenha uma entrada vazia. Essa otimização utiliza a memória de modo mais eficiente, pois as escritas multipalavras normalmente são mais rápidas do que as escritas realizadas uma palavra de cada vez. Skadron e Clark (1997) descobriram que aproximadamente 5-10% do desempenho era perdido devido a stalls em um buffer de escrita de quatro entradas.

A otimização também reduz os stalls devido ao fato de o buffer de escrita estar cheio. A Figura 2.7 mostra um buffer de escrita com e sem a mesclagem da escrita. Suponha que tenhamos quatro entradas no buffer de escrita e que cada entrada possa manter quatro palavras de 64 bits. Sem essa otimização, quatro armazenamentos nos endereços sequenciais preencheriam o buffer em uma palavra por entrada, embora essas quatro palavras, quando mescladas, caibam exatamente dentro de uma única entrada do buffer de escrita.

Observe que os registradores do dispositivo de entrada/saída são então mapeados para o espaço de endereços físico. Esses endereços de E/S não podem permitir a mesclagem da escrita, pois registradores de E/S separados podem não atuar como um array de palavras

Endereço de escrita	V		V		V		V	
100	1	Mem[100]	0		0		0	
108	1	Mem[108]	0		0		0	
116	1	Mem[116]	0		0		0	
124	1	Mem[124]	0		0		0	

Endereço de escrita	V		V		V		V	
100	1	Mem[100]	1	Mem[108]	1	Mem[116]	1	Mem[124]
	0		0		0		0	
	0		0		0		0	
	0		0		0		0	

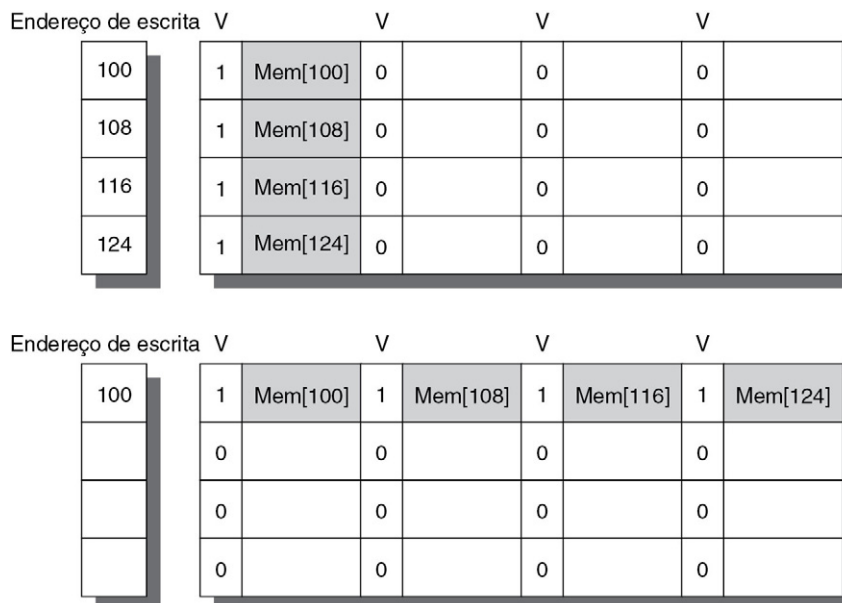



FIGURA 2.7 Para ilustrar a mesclagem da escrita, o buffer de escrita de cima não a utiliza, enquanto o buffer de escrita de baixo a utiliza.

As quatro escritas são mescladas em uma única entrada de buffer com mesclagem de escrita; sem ela, o buffer fica cheio, embora 3/4 de cada entrada sejam desperdiçados. O buffer possui quatro entradas, e cada entrada mantém quatro palavras de 64 bits. O endereço para cada entrada está à esquerda, com um bit de válido (V) indicando se os próximos oito bytes sequenciais nessa entrada são ocupados. (Sem a mesclagem da escrita, as palavras à direita na parte superior da figura não seriam usadas para instruções que escrevessem múltiplas palavras ao mesmo tempo.)

na memória. Por exemplo, eles podem exigir um endereço e uma palavra de dados por registrador, em vez de escritas multipalavras usando um único endereço. Esses efeitos colaterais costumam ser implementados marcando as páginas como requerendo escrita sem mesclagem pelas caches.

Oitava otimização: otimizações de compilador para reduzir a taxa de falta

Até aqui, nossas técnicas têm exigido a mudança do hardware. Essa próxima técnica reduz as taxas de falta sem quaisquer mudanças no hardware.

Essa redução mágica vem do software otimizado — a solução favorita do projetista de hardware! A diferença de desempenho cada vez maior entre os processadores e a memória principal tem inspirado os projetistas de compiladores a investigar a hierarquia de memória para ver se as otimizações em tempo de compilação podem melhorar o desempenho. Mais uma vez, a pesquisa está dividida entre as melhorias nas faltas de instrução e as melhorias nas faltas de dados. As otimizações apresentadas a seguir são encontradas em muitos compiladores modernos.

Permuta de loop

Alguns programas possuem loops aninhados que acessam dados na memória na ordem não sequencial. Simplesmente trocar o aninhamento dos loops pode fazer o código acessar os dados na ordem em que são armazenados. Considerando que os arrays não cabem na cache, essa técnica reduz as faltas, melhorando a localidade espacial; a reordenação maximiza o uso de dados em um bloco de cache antes que eles sejam descartados. Por exemplo, se *x* for um array bidimensional de tamanho [5.000,100] alocado de modo que x[i,j] e x[i,j + 1] sejam adjacentes (uma ordem chamada *ordem principal de linha*, já que o array é organizado em linhas), então os códigos a seguir mostram como os acessos podem ser otimizados:

```
/* Antes */
para (j = 0; j < 100; j = j+1)
      for (i = 0; i < 5.000; i = i+1)
            x[i][j] = 2 * x[i][j];
/* Depois */
para (i = 0; i < 5.000; i = i+1)
     for (j = 0; j < 100; j = j+1)
           x[i][j] = 2 * x[i][j];
```

O código original saltaria pela memória em trechos de 100 palavras, enquanto a versão revisada acessa todas as palavras em um bloco de cache antes de passar para o bloco seguinte. Essa otimização melhora o desempenho da cache sem afetar o número de instruções executadas.

Bloqueio

Essa otimização melhora a localidade temporal para reduzir as faltas. Novamente, estamos lidando com múltiplos arrays, com alguns arrays acessados por linhas e outros por colunas. Armazenar os arrays linha por linha (*ordem principal de linha*) ou coluna por coluna (*ordem principal de coluna*) não resolve o problema, pois linhas e colunas são usadas em cada iteração do loop. Esses acessos ortogonais significam que transformações como permuta de loop ainda possuem muito espaço para melhoria.

Em vez de operar sobre linhas ou colunas inteiras de um array, os algoritmos bloqueados operam sobre submatrizes ou *blocos*. O objetivo é maximizar os acessos aos dados carregados

na cache antes que eles sejam substituídos. O exemplo de código a seguir, que realiza a multiplicação de matriz, ajuda a motivar a otimização:

```
/* Antes */
para (i = 0; i < N; i = i+1)
      para (j = 0; j < N; j = j+1)
            {r = 0;
             para (k = 0; k < N; k = k + 1)
                  r = r + y[i][k]*z[k][j];
             x[i][j] = r;
            };
```

Os dois loops interiores leem todos os elementos *N* por *N* de *z*, leem os mesmos *N* elementos em uma linha de *y* repetidamente e escrevem uma linha de *N* elementos de *x*. A Figura 2.8 apresenta um instantâneo dos acessos aos três arrays. O tom escuro indica acesso recente, o tom claro indica acesso mais antigo e o branco significa ainda não acessado.

O número de faltas de capacidade depende claramente de *N* e do tamanho da cache. Se ele puder manter todas as três matrizes *N* por *N*, tudo está bem, desde que não existam conflitos de cache. Se a cache puder manter uma matriz *N* por *N* e uma linha de *N*, pelo menos a *i*-ésima linha de *y* e o array *z* podem permanecer na cache. Menos do que isso e poderão ocorrer faltas para *x* e *z*. No pior dos casos, haverá $2N^3 + N^2$ palavras de memória acessadas para N^3 operações.

Para garantir que os elementos acessados podem caber na cache, o código original é mudado para calcular em uma submatriz de tamanho *B* por *B*. Dois loops internos agora calculam em passos de tamanho *B*, em vez do tamanho completo de *x* e *z*. *B* é chamado de *fator de bloqueio* (considere que *x* é inicializado com zero).

```
/* Depois */
para (jj = 0; jj < N; jj = jj+B)
para (kk = 0; kk < N; kk = kk+B)
para (i = 0; i < N; i = i+1)
      para (j = jj; j < min(jj+B,N); j = j+1)
          {r = 0;
           para (k = kk; k < min(kk+B,N); k = k + 1)
               r = r + y[i][k]*z[k][j];
           x[i][j] = x[i][j] + r;
          };
```

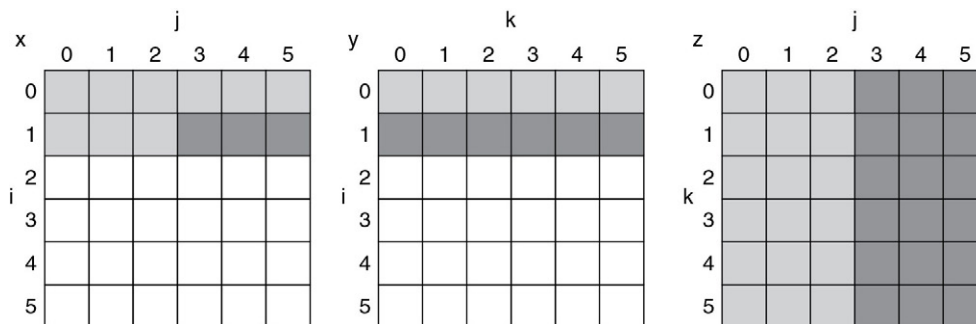



FIGURA 2.8 Um instantâneo dos três arrays *x*, *y* e *z* quando $N = 6$ e $i = 1$.

Os acessos, distribuídos no tempo, aos elementos do array são indicados pelo tom: branco significa ainda não tocado, claro significa acessos mais antigos e escuro significa acessos mais recentes. Em comparação com a Figura 2.9, os elementos de *y* e *z* são lidos repetidamente para calcular novos elementos de *x*. As variáveis *i*, *j* e *k* aparecem ao longo das linhas ou colunas usadas para acessar os arrays.

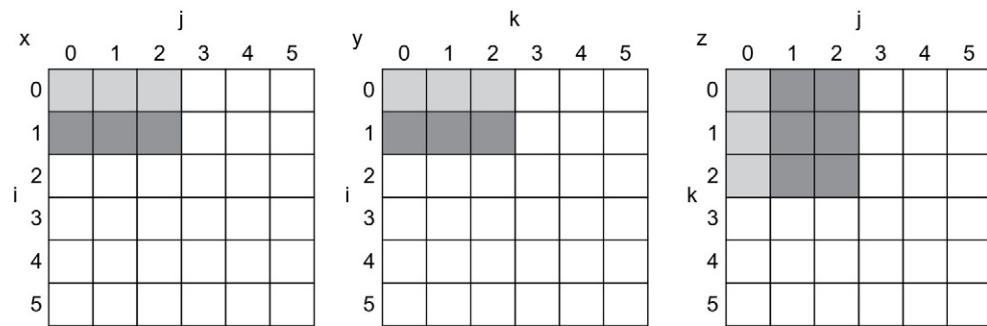



FIGURA 2.9 Acessos, distribuídos no tempo, aos arrays *x*, *y* e *z* quando $B = 3$.
Observe, em comparação com a Figura 2.8, o número menor de elementos acessados.

A Figura 2.9 ilustra os acessos aos três arrays usando o bloqueio. Vendo apenas as perdas de capacidade, o número total de palavras acessadas da memória é $2N^3/B + N^2$. Esse total é uma melhoria por um fator de *B*. Logo, o bloqueio explora uma combinação de localidade espacial e temporal, pois *y* se beneficia com a localidade espacial e *z* se beneficia com a localidade temporal.

Embora tenhamos visado reduzir as faltas de cache, o bloqueio também pode ser usado para ajudar na alocação de registradores. Selecionando um pequeno tamanho de bloqueio, de modo que o bloco seja mantido nos registradores, podemos minimizar o número de carregamentos e armazenamentos no programa.

Como veremos na Seção 4.8 do Capítulo 4, o bloqueio de cache é absolutamente necessário para obter bom desempenho de processadores baseados em cache, executando aplicações que usam matrizes como estrutura de dados primária.

Nona otimização: a pré-busca de pelo hardware das instruções e dados para reduzir a penalidade de falta ou a taxa de falta

As caches de não bloqueio reduzem efetivamente a penalidade de falta, sobrepondo a execução com o acesso à memória. Outra técnica é fazer a pré-busca (*prefetch*) dos itens antes que o processador os solicite. Tanto as instruções quanto os dados podem ter sua busca antecipada, seja diretamente nas caches, seja diretamente em um buffer externo, que pode ser acessado mais rapidamente do que a memória principal.

Frequentemente, a pré-busca de instrução é feita no hardware fora da cache. Em geral, o processador apanha dois blocos em uma falta: o bloco solicitado e o próximo bloco consecutivo. O bloco solicitado é colocado na cache de instruções, e o bloco cuja busca foi antecipada é colocado no buffer do fluxo de instruções. Se o bloco solicitado estiver presente no buffer do fluxo de instruções, a solicitação de cache original será cancelada, o bloco será lido do buffer de fluxo e a próxima solicitação de pré-busca será emitida.

Uma técnica semelhante pode ser aplicada aos acessos a dados (Jouppi, 1990). Palacharla e Kessler (1994) examinaram um conjunto de programas científicos e consideraram múltiplos buffers de fluxo que poderiam tratar tanto instruções como dados. Eles descobriram que oito buffers de fluxo poderiam capturar 50-70% de todas as faltas de um processador com duas caches de 64 KB associativas por conjunto com quatro vias, um para instruções e os outros para dados.

O Intel Core i7 pode realizar a pré-busca de dados no L1 e L2 com o caso de pré-busca mais comum sendo o acesso à próxima linha. Alguns processadores anteriores da Intel usavam uma pré-busca mais agressiva, mas isso resultou em desempenho reduzido para algumas aplicações, fazendo com que alguns usuários sofisticados desativassem o recurso.

A Figura 2.10 mostra a melhoria de desempenho geral para um subconjunto dos programas SPEC2000 quando a pré-busca de hardware está ativada. Observe que essa figura inclui apenas dois de 12 programas inteiros, enquanto inclui a maioria dos programas SPEC de ponto flutuante.

A pré-busca conta com o uso de uma largura de banda de memória que, de outra forma, seria inutilizada, mas, se interferir nas perdas de demanda, pode realmente reduzir o desempenho. A ajuda de compiladores pode reduzir a pré-busca inútil. Quando a pré-busca funciona bem, seu impacto sobre a consumo de energia é irrelevante. Quando dados pré-obtidos não são usados ou dados úteis estão fora do lugar, a pré-busca pode ter um impacto muito negativo no consumo de energia.

Décima otimização: pré-busca controlada por compilador para reduzir a penalidade de falta ou a taxa de falta

Uma alternativa à pré-busca de hardware é que o compilador insira instruções de pré-busca para solicitar dados antes que o processador precise deles. Existem dois tipos de pré-busca:

- A *pré-busca de registrador*, que carrega o valor em um registrador.
- A *pré-busca de cache*, que carrega os dados apenas na cache, e não no registrador.

Qualquer uma delas pode ser *com falta* ou *sem falta*, ou seja, o endereço causa ou não uma exceção para faltas de endereço virtuais e violações de proteção. Usando essa terminologia, uma instrução de carregamento normal poderia ser considerada uma "instrução de pré-busca de registrador com falta". As pré-buscas sem falta simplesmente se transformarão em no-ops se normalmente resultarem em uma exceção, que é o que queremos.

A pré-busca mais efetiva é "semanticamente invisível" a um programa: ela não muda o conteúdo dos registradores e da memória, *e* não pode causar faltas de memória virtual. Hoje a maioria dos processadores oferece pré-buscas de cache sem falta. Esta seção considera a pré-busca de cache sem falta, também chamada *pré-busca sem vínculo*.

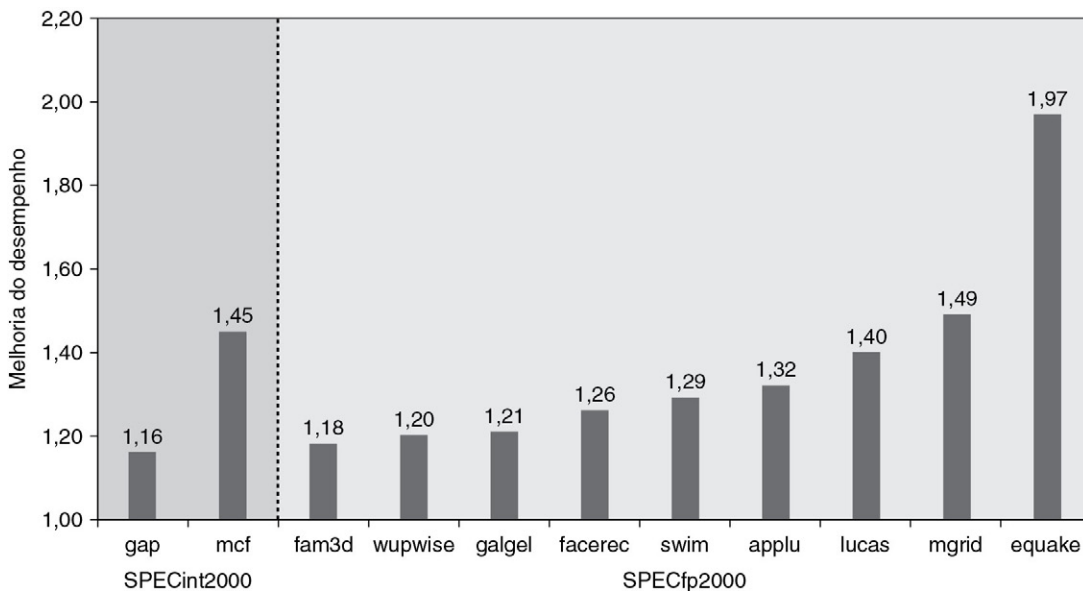



FIGURA 2.10 Ganho de velocidade devido à pré-busca de hardware no Intel Pentium 4 com a pré-busca de hardware ativada para dois dos 12 benchmarks SPECint2000 e nove dos 14 benchmarks SPECfp2000. Somente os programas que se beneficiam mais com a pré-busca são mostrados; a pré-busca agiliza os 15 benchmarks SPEC restantes em menos de 15% (Singhal, 2004).

A pré-busca só faz sentido se o processador puder prosseguir enquanto realiza a pré-busca dos dados, ou seja, as caches não param, mas continuam a fornecer instruções e dados enquanto esperam que os dados cujas buscas foram antecipadas retornem. Como era de esperar, a cache de dados para esses computadores normalmente é sem bloqueio.

Assim como na pré-busca controlada pelo hardware, o objetivo é sobrepor a execução com a pré-busca de dados. Os loops são os alvos importantes, pois servem para otimizações de pré-busca. Se a penalidade de falta for pequena, o compilador simplesmente desdobrará o loop uma ou duas vezes e escalonará as pré-buscas com a execução. Se a penalidade de falta for grande, ele usará o pipelining de software (Apêndice H) ou desdobrará muitas vezes para realizar a pré-busca de dados para uma iteração futura.

Entretanto, emitir instruções de pré-busca contrai um overhead de instrução, de modo que os compiladores precisam tomar cuidado para garantir que tais overheads não sejam superiores aos benefícios. Concentrando-se em referências que provavelmente serão faltas de cache, os programas podem evitar pré-buscas desnecessárias enquanto melhoram bastante o tempo médio de acesso à memória.

Exemplo Para o código a seguir, determine quais acessos provavelmente causarão faltas de cache de dados. Em seguida, insira instruções de pré-busca para reduzir as faltas. Finalmente, calcule o número de instruções de pré-busca executadas e as faltas evitadas pela pré-busca. Vamos supor que tenhamos uma cache de dados mapeado diretamente de 8 KB com blocos de 16 bytes e ela seja uma cache write-back que realiza alocação de escrita. Os elementos de *a* e *b* possuem oito bytes, pois são arrays de ponto flutuante de precisão dupla. Existem três linhas e 100 colunas para *a* e 101 linhas e três colunas para *b*. Vamos supor também que eles não estejam na cache no início do programa

```
para (i = 0; i < 3; i = i+1)
      para (j = 0; j < 100; j = j+1)
             a[i][j] = b[j][0] * b[j+1][0];
```

Resposta O compilador primeiro determinará quais acessos provavelmente causarão faltas de cache; caso contrário, perderemos tempo emitindo instruções de pré-busca para dados que seriam acertos. Os elementos de *a* são escritos na ordem em que são armazenados na memória, de modo que *a* se beneficiará com a proximidade espacial: os valores pares de *j* serão faltas, e os valores ímpares serão acertos. Como *a* possui três linhas e 100 colunas, seus acessos levarão a $3 \times (100/2)$, ou 150 faltas.

O array *b* não se beneficia com a proximidade espacial, pois os acessos não estão na ordem em que são armazenados. O array *b* se beneficia duas vezes da localidade temporal: os mesmos elementos são acessados para cada iteração de *i*, e cada iteração de *j* usa o mesmo valor de *b* que a última iteração. Ignorando faltas de conflito em potencial, as faltas devidas a *b* serão para b[j+1][0] acessos quando i = 0, e também o primeiro acesso a b[j][0] quando j = 0. Como *j* vai de 0 a 99 quando i = 0, os acessos a *b* levam a 100 + 1 ou 101 faltas.

Assim, esse loop perderá a cache de dados aproximadamente 150 vezes para *a* mais 101 vezes para *b* ou 251 faltas.

Para simplificar nossa otimização, não nos preocuparemos com a pré-busca dos primeiros acessos do loop. Eles já podem estar na cache ou pagaremos a penalidade de falta dos primeiros poucos elementos de *a* ou *b*. Também não nos preocuparemos em suprimir as pré-buscas ao final do loop, que tentam buscar previamente além do final de *a* (a[i][100] … a[i][106]) e o final de *b* (b[101][0] … b[107][0]). Se essas pré-buscas fossem com falta,

não poderíamos ter esse luxo. Vamos considerar que a penalidade de falta é tão grande que precisamos começar a fazer a pré-busca pelo menos sete iterações à frente (em outras palavras, consideramos que a pré-busca não tem benefício até a oitava iteração). Sublinhamos as mudanças feitas no código anterior, necessárias para realizar a pré-busca.

```
para (j = 0; j < 100; j = j+1) {
      prefetch(b[j+7][0]);
      /* b(j,0) para 7 iterações depois */
      prefetch(a[0][j+7]);
      /* a(0,j) para 7 iterações depois */
      a[0][j] = b[j][0] * b[j+1][0];};
para (i = 1; i < 3; i = i+1)
      para (j = 0; j < 100; j = j+1) {
            prefetch(a[i][j+7]);
            /* a(i,j) para +7 iterações */
            a[i][j] = b[j][0] * b[j+1][0];}
```

Esse código revisado realiza a pré-busca de a[i][7] até a[i][99] e de b[7][0] até b[100][0], reduzindo o número de faltas de não pré-busca para

- Sete faltas para os elementos b[0][0], b[1][0], ..., b[6][0] no primeiro loop
- Quatro faltas ([7/2]) para os elementos a[0][0], a[0][1], ..., a[0][6] no primeiro loop (a proximidade espacial reduz as faltas para uma por bloco de cache de 16 bytes)
- Quatro faltas ([7/2]) para os elementos a[1][0], a[1][1], ..., a[1][6] no segundo loop.
- Quatro faltas ([7/2]) para os elementos a[2][0], a[2][1], ..., a[2][6] no segundo loop

ou um total de 19 faltas de não pré-busca. O custo de evitar 232 faltas de cache é a execução de 400 instruções de pré-busca, provavelmente uma boa troca.

Exemplo Calcule o tempo economizado no exemplo anterior. Ignore faltas da cache de instrução e considere que não existem faltas por conflito ou capacidade na cache de dados. Suponha que as pré-buscas possam se sobrepor umas às outras com faltas de cache, transferindo, portanto, na largura de banda máxima da memória. Aqui estão os principais tempos de loop ignorando as faltas de cache: o loop original leva sete ciclos de clock por iteração, o primeiro loop de pré-busca leva nove ciclos de clock por iteração e o segundo loop de pré-busca leva oito ciclos de clock por iteração (incluindo o overhead do loop for externo). Uma falta leva 100 ciclos de clock.

Resposta O loop original duplamente aninhado executa a multiplicação 3×100 ou 300 vezes. Como o loop leva sete ciclos de clock por iteração, o total é de 300×7 ou 2.100 ciclos de clock mais as faltas de cache. As faltas de cache aumentam 251×100 ou 25.100 ciclos de clock, gerando um total de 27.200 ciclos de clock. O primeiro loop de pré-busca se repete 100 vezes; a nove ciclos de clock por iteração, o total é de 900 ciclos de clock mais as faltas de cache. Elas aumentam 11×100 ou 1.100 ciclos de clock para as faltas de cache, gerando um total de 2.000. O segundo loop é executado 2×100 ou 200 vezes, e a nove ciclos de clock por iteração; isso leva 1.600 ciclos de clock mais 8×100 ou 800 ciclos de clock para as faltas de cache. Isso gera um total de 2.400 ciclos de clock. Do exemplo anterior, sabemos que esse código executa 400 instruções de pré-busca durante os 2.000 + 2.400 ou 4.400 ciclos de clock para executar esses dois loops. Se presumirmos que as pré-buscas são completamente sobrepostas com o restante da execução, então o código da pré-busca é 27.200/4.400 ou 6,2 vezes mais rápido.

Embora as otimizações de array sejam fáceis de entender, os programas modernos provavelmente utilizam ponteiros. Luk e Mowry (1999) demonstraram que a pré-busca baseada em compilador às vezes também pode ser estendida para ponteiros. Dos 10 programas com estruturas de dados recursivas, a pré-busca de todos os ponteiros quando um nó é visitado melhorou o desempenho em 4-31% na metade dos programas. Por outro lado, os programas restantes ainda estavam dentro de 2% de seu desempenho original. A questão envolve tanto se as pré-buscas são para dados já na cache quanto se elas ocorrem cedo o suficiente para os dados chegarem quando forem necessários.

Muitos processadores suportam instruções para pré-busca de cache e, muitas vezes, processadores de alto nível (como o Intel Core i7) também realizam algum tipo de pré-busca automática no hardware.

Resumo de otimização de cache

As técnicas para melhorar o tempo de acerto, a largura de banda, a penalidade de falta e a taxa de falta geralmente afetam os outros componentes da equação de acesso médio à memória, além da complexidade da hierarquia de memória. A Figura 2.11 resume essas técnicas e estima o impacto sobre a complexidade, com + significando que a técnica melhora o fator, – significando que ela prejudica esse fator, e um espaço significando que ela não tem impacto. Geralmente, nenhuma técnica ajuda mais de uma categoria.

Técnica	Tempo de acerto	Largura de banda	Penalidade de falta	Taxa de falta	Custo/ complexidade do hardware	Comentário
Caches pequenas e simples	+			–	0	Trivial; muito usada
Caches de previsão de via	+				1	Usada no Pentium 4
Acesso à cache em pipeline	–	+			1	Muito usada
Caches sem bloqueio		+	+		3	Muito usada
Caches em banco		+			1	Usada no L2 do Opteron e Niagara
Palavra crítica primeiro e reinício antecipado			+		2	Muito usada
Write buffer merge			+		1	Muito usada com write-through
Técnicas de compilador para reduzir faltas de cache				+	0	O software é um desafio; alguns computadores possuem opção do compilador
Pré-busca de hardware de instruções e dados			+	+	2 para instrução, 3 para dados	Muitas instruções de pré-busca; dados de pré-busca do Opteron e Pentium 4
Pré-busca controlada pelo compilador			+	+	3	Precisa de cache sem bloqueio; possível overhead de instrução; em muitas CPUs

FIGURA 2.11 Resumo das 10 otimizações de cache avançadas mostrando o impacto sobre o desempenho da cache e a complexidade.
Embora geralmente uma técnica só ajude um fator, a pré-busca pode reduzir as faltas se for feita suficientemente cedo; se isso não ocorrer, ela pode reduzir a penalidade de falta. + significa que a técnica melhora o fator, – significa que ela prejudica esse fator, e um espaço significa que ela não tem impacto. A medida de complexidade é subjetiva, com 0 sendo o mais fácil e 3 sendo um desafio.

2.3 TECNOLOGIA DE MEMÓRIA E OTIMIZAÇÕES

[…] o único desenvolvimento isolado que colocou os computadores na linha foi a invenção de uma forma confiável de memória, a saber, a memória de núcleo […] Seu custo foi razoável, ela era confiável e, por ser confiável, poderia se tornar grande com o passar do tempo. (p. 209)

Maurice Wilkes
Memoirs of a Computer Pioneer (1985)

A memória principal é o próximo nível abaixo na hierarquia. A memória principal satisfaz as demandas das caches e serve de interface de E/S, pois é o destino da entrada e também a origem da saída. As medidas de desempenho da memória principal enfatizam tanto a latência quanto a largura de banda. Tradicionalmente, a latência da memória principal (que afeta a penalidade de falta de cache) é a principal preocupação da cache, enquanto a largura de banda da memória principal é a principal preocupação dos multiprocessadores e da E/S.

Embora as caches se beneficiem da memória de baixa latência, geralmente é mais fácil melhorar a largura de banda da memória com novas organizações do que reduzir a latência. A popularidade das caches de segundo nível e de seus tamanhos de bloco maiores torna a largura de banda da memória principal importante também para as caches. Na verdade, os projetistas de cache aumentam o tamanho de bloco para tirar proveito da largura de banda com alto desempenho.

As seções anteriores descrevem o que pode ser feito com a organização da cache para reduzir essa diferença de desempenho processador-DRAM, mas simplesmente tornar as caches maiores ou acrescentar mais níveis de caches não elimina a diferença. Inovações na memória principal também são necessárias.

No passado, a inovação era o modo de organizar os muitos chips de DRAM que compunham a memória principal, assim como os múltiplos bancos de memória. Uma largura de banda maior está disponível quando se usam bancos de memória, tornando a memória e seu barramento mais largos ou fazendo ambos. Ironicamente, à medida que a capacidade por chip de memória aumenta, existem menos chips no sistema de memória do mesmo tamanho, reduzindo as possibilidades para sistemas de memória mais largos com a mesma capacidade.

Para permitir que os sistemas de memória acompanhem as demandas de largura de banda dos processadores modernos, as inovações de memória começaram a acontecer dentro dos próprios chips de DRAM. Esta seção descreve a tecnologia dentro dos chips de memória e essas organizações inovadoras, internas. Antes de descrever as tecnologias e as opções, vamos examinar as medidas de desempenho.

Com a introdução das memórias de transferência pelo modo burst, hoje amplamente usadas em memórias Flash e DRAM, a latência da memória é calculada usando duas medidas: tempo de acesso e tempo de ciclo. O *tempo de acesso* é o tempo entre a solicitação de uma leitura e a chegada da palavra desejada, enquanto *tempo de ciclo* é o tempo mínimo entre as solicitações e a memória.

Quase todos os computadores desktops ou servidores utilizam, desde 1975, DRAMs para a memória principal e quase todos utilizam SRAMs para cache, com 1-3 níveis integrados no chip do processador com a CPU. Em PMDs, a tecnologia de memória muitas vezes equilibra consumo de energia e velocidade com sistemas de alto nível, usando tecnologia de memória rápida e com grande largura de banda.

Tecnologia de SRAM

A primeira letra de SRAM significa *estática* (*static*, em inglês). A natureza dinâmica dos circuitos na DRAM exige que os dados sejam escritos de volta após serem lidos — daí a diferença entre o tempo de acesso e o tempo de ciclo, além da necessidade de atualização (*refresh*). As SRAMs não precisam ser atualizadas e, portanto, o tempo de acesso é muito próximo do tempo de ciclo. As SRAMs normalmente utilizam seis transistores por bit para impedir que a informação seja modificada quando lida. A SRAM só precisa de um mínimo de energia para reter a carga no modo de stand-by.

No princípio, a maioria dos sistemas de desktops e servidores usava chip de SRAM para suas caches primárias, secundárias ou terciárias. Hoje, os três níveis de caches são integrados no chip do processador. Atualmente, os maiores caches de terceiro nível no chip são de 12 MB, enquanto o sistema de memória para tal processador provavelmente terá de 4-16 GB de DRAM. Em geral, os tempos de acesso para grandes caches de terceiro nível no chip são 2-4 vezes os de uma cache de segundo nível, que ainda é 3-5 vezes mais rápido do que acessar uma memória DRAM.

Tecnologia de DRAM

À medida que as primeiras DRAMs cresciam em capacidade, o custo de um pacote com todas as linhas de endereço necessárias se tornava um problema. A solução foi multiplexar as linhas de endereço, reduzindo assim o número de pinos de endereço ao meio. A Figura 2.12 mostra a organização básica da DRAM. Metade do endereço é enviada primeiro, algo chamado RAS — Row Access Strobe. A outra metade do endereço, enviada durante o CAS — Column Access Strobe —, vem depois. Esses nomes vêm da organização interna do chip, pois a memória é organizada como uma matriz retangular endereçada por linhas e colunas.

Um requisito adicional da DRAM deriva da propriedade indicada pela primeira letra, *D*, de *dinâmica*. Para compactar mais bits por chips, as DRAMs utilizam apenas um único transistor para armazenar um bit. A leitura desse bit destrói a informação, de modo que ela precisa ser restaurada. Esse é um motivo pelo qual o tempo de ciclo da DRAM é muito maior que o tempo de acesso. Além disso, para evitar perda de informações quando um bit não é lido ou escrito, o bit precisa ser "restaurado" periodicamente (refresh). Felizmente, todos os bits em uma linha podem ser renovados simultaneamente apenas pela leitura

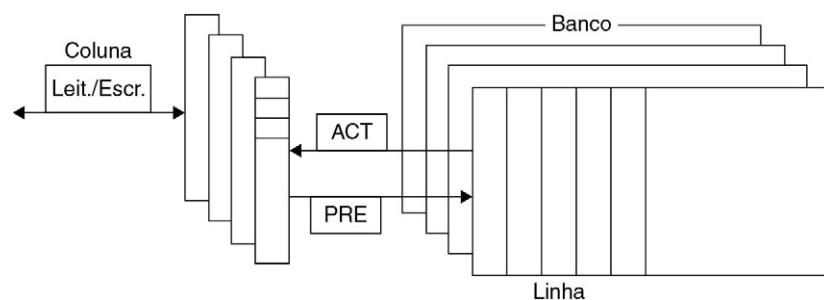



FIGURA 2.12 **Organização interna de uma DRAM.**
As DRAMs modernas são organizadas em bancos, em geral quatro por DDR3. Cada banco consiste em uma série de linhas. Enviar um comando PRE (pré-carregamento) abre ou fecha um banco. Um endereço de linha é enviado com um ACT (ativar), que faz com que a linha seja transferida para um buffer. Quando a linha está no buffer, ela pode ser transferida por endereços de coluna sucessivos em qualquer largura da DRAM (geralmente 4, 8 ou 16 bits em DDR3) ou especificando uma transferência de bloco e o endereço de início. Cada comando, assim como as transferências de bloco, é sincronizado com um clock.

dessa linha. Logo, cada DRAM do sistema de memória precisa acessar cada linha dentro de certa janela de tempo, como 8 ms. Os controladores de memória incluem o hardware para refresh das DRAMs periodicamente.

Esse requisito significa que o sistema de memória está ocasionalmente indisponível, pois está enviando um sinal que diz a cada chip para ser restaurado. O tempo para um refresh normalmente é um acesso completo à memória (RAS e CAS) para cada linha da DRAM. Como a matriz de memória em uma DRAM é conceitualmente quadrada, em geral o número de etapas em um refresh é a raiz quadrada da capacidade da DRAM. Os projetistas de DRAM tentam manter o tempo gasto na restauração menor que 5% do tempo total.

Até aqui, apresentamos a memória principal como se ela operasse como um trem suíço, entregando suas mercadorias de modo consistente, exatamente de acordo com o horário. O refresh contradiz essa analogia, pois alguns acessos levam muito mais tempo do que outros. Assim, o refresh é outro motivo para a variabilidade da latência da memória e, portanto, da penalidade de falta da cache.

Amdahl sugeriu uma regra prática de que a capacidade da memória deverá crescer linearmente com a velocidade do processador para manter um sistema equilibrado, de modo que um processador de 1.000 MIPS deverá ter 1.000 MB de memória. Os projetistas de processador contam com as DRAMs para atender a essa demanda: no passado, eles esperavam uma melhoria que quadruplicasse a capacidade a cada três anos ou 55% por ano. Infelizmente, o desempenho das DRAMs está crescendo em uma taxa muito mais lenta. A Figura 2.13 mostra a melhora de desempenho no tempo de acesso da linha, que está relacionado com a latência, de cerca de 5% por ano. O CAS, ou tempo de transferência de dados, relacionado com a largura de banda está crescendo em mais de duas vezes essa taxa.

			Row access strobe (RAS)			
Ano de produção	**Tamanho do chip**	**Tipo de DRAM**	**DRAM mais lenta (ns)**	**DRAM mais rápida (ns)**	**Column Access Strobe (CAS)/tempo de transferência de dados (ns)**	**Tempo de ciclo (ns)**
1980	64 K bits	DRAM	180	150	75	250
1983	256 K bits	DRAM	150	120	50	220
1986	1 M bits	DRAM	120	100	25	190
1989	4 M bits	DRAM	100	80	20	165
1992	16 M bits	DRAM	80	60	15	120
1996	64 M bits	SDRAM	70	50	12	110
1998	128 M bits	SDRAM	70	50	10	100
2000	256 M bits	DDR1	65	45	7	90
2002	512 M bits	DDR1	60	40	5	80
2004	1 G bits	DDR2	55	35	5	70
2006	2 G bits	DDR2	50	30	2,5	60
2010	4 G bits	DDR3	36	28	1	37
2012	8 G bits	DDR3	30	24	0,5	31

FIGURA 2.13 Tempos de DRAMs rápidas e lentas a cada geração.

(O tempo de ciclo foi definido na página 83.) A melhoria de desempenho do tempo de acesso de linha é de cerca de 5% por ano. A melhoria por um fator de 2 no acesso à coluna em 1986 acompanhou a troca das DRAMs NMOS para DRAMs CMOS. A introdução de vários modos burst de transferência, em meados dos anos 1990, e as SDRAMs, no final dos anos 1990, complicaram significativamente o cálculo de tempo de acesso para blocos de dados. Vamos discutir isso de modo aprofundado nesta seção quando falarmos sobre tempo de acesso e potência da SDRAM. O lançamento dos projetos DDR4 está previsto para o segundo semestre de 2013. Vamos discutir essas diversas formas de DRAM nas próximas páginas.

Embora estejamos falando de chips individuais, as DRAMs normalmente são vendidas em pequenas placas, chamadas *módulos de memória em linha dupla* (Dual Inline Memory Modules — DIMMs). Os DIMMs normalmente contêm 4-16 DRAMs e são organizados para ter oito bytes de largura (+ ECC) para sistemas desktops.

Além do empacotamento do DIMM e das novas interfaces para melhorar o tempo de transferência de dados, discutidos nas próximas subseções, a maior mudança nas DRAMs tem sido uma redução no crescimento da capacidade. As DRAMs obedeceram à lei de Moore por 20 anos, gerando um novo chip com capacidade quadruplicada a cada três anos. Devido aos desafios de fabricação de uma DRAM de bit único, novos chips só dobraram de capacidade a cada dois anos a partir de 1998. Em 2006, o ritmo foi reduzido ainda mais, com o período de 2006 a 2010 testemunhando somente uma duplicação na capacidade.

Melhorando o desempenho da memória dentro de um chip de DRAM

À medida que a lei de Moore continua a fornecer mais transistores e a diferença entre processador-memória aumenta a pressão sobre o desempenho da memória, as ideias da seção anterior se encaminharam para dentro do chip da DRAM. Geralmente, a inovação tem levado a maior largura de banda, às vezes ao custo da maior latência. Esta subseção apresenta técnicas que tiram proveito da natureza das DRAMs.

Como já mencionamos, um acesso à DRAM é dividido em acessos de linha e acessos de coluna. As DRAMs precisam colocar uma linha de bits no buffer dentro da DRAM para o acesso de coluna, e essa linha normalmente é a raiz quadrada do tamanho da DRAM — por exemplo, 2 Kb para uma DRAM de 4 Mb. Conforme as DRAMs cresceram, foram adicionadas estruturas adicionais e diversas oportunidades para aumentar a largura de banda.

Primeiramente, as DRAMs adicionaram a temporização de sinais que permitem acessos repetidos ao buffer de linha sem outro tempo de acesso à linha. Tal buffer vem naturalmente quando cada array colocará de 1.024 a 4.096 bits no buffer para cada acesso. Inicialmente, endereços separados de coluna tinham de ser enviados para cada transferência com um atraso depois de cada novo conjunto de endereços de coluna.

Originalmente, as DRAMs tinham uma interface assíncrona para o controlador de memória e, portanto, cada transferência envolvia o overhead para sincronizar com o controlador. A segunda principal mudança foi acrescentar um sinal de clock à interface DRAM, de modo que as transferências repetidas não sofram desse overhead. *DRAM síncrona* (SDRAM) é o nome dessa otimização. Normalmente, as SDRAMs também tinham um registrador programável para manter o número de bytes solicitados e, portanto, podiam enviar muitos bytes por vários ciclos por solicitação. Em geral, oito ou mais transferências de 16 bits podem ocorrer sem enviar novos endereços colocando a DRAM em modo de explosão. Esse modo, que suporta transferências de palavra crítica em primeiro lugar, é o único em que os picos de largura de banda mostrados na Figura 2.14 podem ser alcançados.

Em terceiro lugar, para superar o problema de obter um grande fluxo de bits da memória sem ter de tornar o sistema de memória muito grande, conforme a densidade do sistema de memória aumenta, as DRAMs se tornaram mais largas. Inicialmente, elas ofereciam um modo de transferência de 4 bits. Em 2010, as DRAMs DDR2 e DDR3 tinham barramentos de até 16 bits.

A quarta inovação importante da DRAM para aumentar a largura de banda é transferir dados tanto na borda de subida quanto na borda de descida no sinal de clock da DRAM, dobrando assim a taxa de dados de pico. Essa otimização é chamada *taxa de dados dupla* (Double Data Rate — DDR).

Padrão	Taxa de clock (MHz)	M transferências por segundo	Nome DRAM	MB/s/DIMM	Nome DIMM
DDR	133	266	DDR266	2.128	PC2100
DDR	150	300	DDR300	2.400	PC2400
DDR	200	400	DDR400	3.200	PC3200
DDR2	266	533	DDR2-533	4.264	PC4300
DDR2	333	667	DDR2-667	5.336	PC5300
DDR2	400	800	DDR2-800	6.400	PC6400
DDR3	533	1.066	DDR3-1066	8.528	PC8500
DDR3	666	1.333	DDR3-1333	10.664	PC10700
DDR3	800	1.600	DDR3-1600	12.800	PC12800
DDR4	1.066-1.600	2.133-3.200	DDR4-3200	17.056-25.600	PC25600

FIGURA 2.14 Taxas de clock, largura de banda e nomes de DRAMs e DIMMs DDR em 2010.
Observe o relacionamento numérico entre as colunas. A terceira coluna é o dobro da segunda, e a quarta usa o número da terceira coluna no nome do chip DRAM. A quinta coluna é oito vezes a terceira coluna, e uma versão arredondada desse número é usada no nome do DIMM. Embora não aparecendo nessa figura, as DDRs também especificam a latência em ciclos de clock. Por exemplo, o DDR3-2000 CL 9 tem latências de 9-9-9-28. O que isso significa? Com um clock de 1 ns (o ciclo de clock é metade da taxa de transferência), isso indica 9 ns para endereços de linha para coluna (tempo RAS), 9 ns para acesso de coluna aos dados (tempo CAS) e um tempo de leitura mínimo de 28 ns. Fechar a linha leva 9 ns para pré-carregamento, mas ocorre somente quando as leituras da linha foram completadas. Em modo de explosão, o pré-carregamento não é necessário até que toda a linha seja lida. O DDR4 será produzido em 2013 e espera-se que atinja taxas de clock de 1.600 MHz em 2014, quando se espera que o DDR5 assuma. Os exercícios exploram esses detalhes.

Para fornecer algumas das vantagens do interleaving, além de ajudar no gerenciamento de energia, as SDRAMs também introduziram os *bancos*, afastando-se de uma única SDRAM para 2-8 blocos (nas DRAMs DDR3 atuais), que podem operar independentemente (já vimos bancos serem usados em caches internas; muitas vezes, eles são usados em grandes memórias principais). Criar múltiplos bancos dentro de uma DRAM efetivamente adiciona outro segmento ao endereço, que agora consiste em número do banco, endereço da linha e endereço da coluna. Quando é enviado um endereço que designa um novo banco, esse banco deve ser aberto, incorrendo em um atraso adicional. O gerenciamento de bancos e buffers de linha é manipulado completamente por interfaces modernas de controle de memória, de modo que, quando um acesso subsequente especifica a mesma linha por um banco aberto, o acesso pode ocorrer rapidamente, enviando somente o endereço da coluna.

Quando as SDRAMs DDR são montadas como DIMMs, são rotuladas pela largura de banda *DIMM* de pico, confusamente. Logo, o nome do DIMM PC2100 vem de 133 MHz $\times$ 2 $\times$ 8 bytes ou 2.100 MB/s. Sustentando a confusão, os próprios chips são rotulados como *número de bits por segundo*, em vez da sua taxa de clock, de modo que um chip DDR de 133 MHz é chamado de DDR266. A Figura 2.14 mostra o relacionamento entre a taxa de clock, as transferências por segundo por chip, nome de chip, largura de banda de DIMM e nome de DIMM.

A DDR agora é uma sequência de padrões. A DDR2 reduz a potência diminuindo a voltagem de 2,5 volts para 1,8 volt e oferece maiores taxas de clock: 266 MHz, 333 MHz e 400 MHz. A DDR3 reduz a voltagem para 1,5 volt e tem velocidade de clock máxima de 800 MHz. A DDR4, prevista para entrar em produção em 2014, diminui a voltagem para 1-1,2 volts e tem taxa de clock máxima esperada de 1.600 MHz. A DDR5 virá em 2014 ou 2015. (Como discutiremos na próxima seção, a GDDR5 é uma RAM gráfica e baseada nas DRAMs DDR3.)

RAMs de dados gráficos

GDRAMs ou GSDRAMs (DRAMs gráficas ou gráficas síncronas) são uma classe especial de DRAMs baseadas nos projetos da SDRAM, mas ajustadas para lidar com as exigências maiores de largura de banda das unidades de processamento gráfico. A GDDR5 é baseada na DDR3, com as primeiras GDDRs baseadas na DDR2. Uma vez que as unidades de processamento gráfico (Graphics Processor Units — GPUs; Cap. 4) requerem mais largura de banda por chip de DRAM do que as CPUs, as GDDRs têm várias diferenças importantes:

1. As GDDRs têm interfaces mais largas: 32 bits contra 4, 8 ou 16 nos projetos atuais.
2. As GDDRs têm taxa máxima de clock nos pinos de dados. Para permitir taxa de transferência maior sem incorrer em problemas de sinalização, em geral as GDRAMS se conectam diretamente à GPU e são conectadas por solda à placa, ao contrário das DRAMs, que costumam ser colocadas em um array barato de DIMMs.

Juntas, essas características permitem às GDDRs rodar com 2-5 vezes a largura de banda por DRAM em comparação às DRAMs DDR3, uma vantagem significativa para suportar as GPUs. Devido à menor distância entre as requisições de memória em um GPU, o modo burst geralmente é menos útil para uma GPU, mas manter múltiplos bancos de memória abertos e gerenciar seu uso melhora a largura de banda efetiva.

Reduzindo o consumo de energia nas SDRAMs

O consumo de energia em chips de memória dinâmica consiste na potência dinâmica usada em uma leitura ou escrita e na potência estática ou de stand-by. As duas dependem da voltagem de operação. Nos SDRAMs DDR3 mais avançados, a voltagem de operação caiu para 1,35-1,5 volt, reduzindo significativamente o consumo de energia em relação às SDRAMs DDR2. A adição dos bancos também reduziu o consumo de energia, uma vez que somente a ilha em um único banco é lida e pré-carregada.

Além dessas mudanças, todas as SDRAMs recentes suportam um modo de *power down*, que é iniciado dizendo à DRAM para ignorar o clock. O modo power down desabilita a SDRAM, exceto pela atualização interna automática (sem a qual entrar no modo power down por mais tempo do que o tempo de atualização vai fazer o conteúdo da memória ser perdido). A Figura 2.15 mostra o consumo de energia em três situações em uma SDRAM

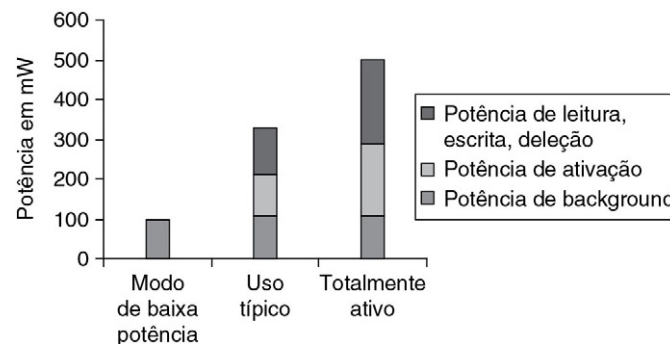



FIGURA 2.15 Consumo de energia para uma SDRAM DDR3 operando sob três condições: modo de baixa potência (shutdown), modo de sistema típico (a DRAM está ativa 30% do tempo para leituras e 15% para escritas) e modo totalmente ativo, em que a DRAM está continuamente lendo ou escrevendo quando não está em pré-carregamento.

Leituras e escritas assumem burts de oito transferências. Esses dados são baseados em um Micron 1,5 V de 2 Gb DDR3-1066.

DDR3 de 2 GB. O atraso exato necessário para retornar do modo de baixa potência depende da SDRAM, mas um tempo típico da atualização do modo de baixa potência é de 200 ciclos de clock. Pode ser necessário tempo adicional para resetar o registrador de modo antes do primeiro comando.

Memória Flash

A memória Flash é um tipo de EEPROM (Electronically Erasable Programmable Read-Only Memory), que normalmente se presta somente à leitura, mas pode ser apagada. A outra propriedade-chave da memória Flash é que ela mantém seu conteúdo sem qualquer alimentação.

A memória Flash é usada como armazenamento de backup nos PMDs do mesmo modo que um disco em um laptop ou servidor. Além disso, uma vez que a maioria dos PMDs tem quantidade limitada de DRAM, a memória Flash também pode agir como um nível da hierarquia de memória, muito mais do que precisaria ser no desktop ou no servidor com uma memória principal que pode ser de 10-100 vezes maior.

A memória Flash usa uma arquitetura muito diferente e tem propriedades diferentes da DRAM padrão. As diferenças mais importantes são:

1. A memória Flash deve ser apagada (por isso o nome Flash, do processo "flash" de apagar) antes que seja sobrescrita, e isso deve ser feito em blocos (em memórias Flash de alta densidade, chamadas NAND Flash, usadas na maioria das aplicações de computador) em vez de bytes ou palavras individuais. Isso significa que, quando dados precisam ser gravados em uma memória Flash, todo um bloco deve ser montado, seja como um bloco de dados novos, seja mesclando os dados a serem gravados e o resto do conteúdo do bloco.
2. A memória Flash é estática (ou seja, ela mantém seu conteúdo mesmo quando não é aplicada energia) e consome significativamente menos energia quando não se está lendo ou gravando (de menos de metade em modo stand-by a zero quando completamente inativa).
3. A memória Flash tem número limitado de ciclos de escrita em qualquer bloco (em geral, pelo menos 100.000). Ao garantir a distribuição uniforme dos blocos escritos por toda a memória, um sistema pode maximizar o tempo de vida de um sistema de memória Flash.
4. Memórias Flash de alta densidade são mais baratas do que a SDRAM, porém são mais caras do que discos: aproximadamente US$ 2/GB para Flash, US$ 20-40/GB para SDRAM e US$ 0,09/GB para discos magnéticos.
5. A memória Flash é muito mais lenta do que a SDRAM, porém muito mais rápida do que um disco. Por exemplo, uma transferência de 256 bytes de uma memória Flash típica de alta densidade leva cerca de 6,5 μs (usando uma transferência em modo de explosão similar, porém mais lenta do que a usada na SDRAM). Uma transferência comparável de uma SDRAM DDR leva cerca de um quarto desse tempo, e para um disco é cerca de 1.000 vezes mais demorada. Para escritas, a diferença é consideravelmente maior, com a SDRAM sendo pelo menos 10 vezes e no máximo 100 vezes mais rápida do que a memória Flash, dependendo das circunstâncias.

As rápidas melhorias na memória Flash de alta densidade na década passada tornaram a tecnologia uma parte viável das hierarquias de memória em dispositivos móveis e também como substitutos dos discos. Conforme a taxa de aumento na densidade da DRAM continua a cair, a memória Flash pode ter um papel cada vez maior nos sistemas

de memória futuros, agindo tanto como um substituto para os discos rígidos quanto como armazenamento intermediário entre a DRAM e o disco.

Aumentando a confiabilidade em sistemas de memória

Caches e memórias principais grandes aumentam significativamente a possibilidade de erros ocorrerem tanto durante o processo de fabricação quanto dinamicamente, principalmente resultantes de raios cósmicos que atingem a célula de memória. Esses erros dinâmicos, que são mudanças no conteúdo de uma célula, e não uma mudança nos circuitos, são chamados *soft errors.* Todas as DRAMs, memórias Flash e muitas SRAMs são fabricadas com linhas adicionais para que pequeno número de defeitos de fabricação possa ser acomodado, programando a substituição de uma linha defeituosa por uma linha adicional. Além de erros de fabricação que podem ser reparados no momento da configuração, também podem ocorrer na operação os *hard errors*, que são mudanças permanentes na operação de uma ou mais células de memória.

Erros dinâmicos podem ser detectados por bits de paridades, detectados e corrigidos pelo uso de códigos de correção de erro (Error Correcting Codes — ECCs). Uma vez que as caches de instrução são somente para leitura, a paridade é o suficiente. Em caches de dados maiores e na memória principal, ECCs são usados para permitir que os erros sejam detectados e corrigidos. A paridade requer somente um bit de overhead para detectar um único erro em uma sequência de bits. Já que um erro multibit não seria detectado com paridade, os números de bits protegidos por um bit de paridade devem ser limitados. Um bit de paridade para 8 bits de dados é uma razão típica. ECCs podem detectar dois erros e corrigir um único erro com um custo de 8 bits de overhead para 64 bits de dados.

Em sistema grandes, a possibilidade de múltiplos erros além da falha completa em um único chip de memória se torna importante. O Chipkill foi lançado pela IBM para solucionar esse problema, e vários sistemas grandes, como servidores IBM e SUN e os Google Clusters, usam essa tecnologia (a Intel chama sua versão de SDDC). Similar em natureza à técnica RAID usada para discos, o Chipkill distribui os dados e informações de ECC para que a falha completa de um único chip de memória possa ser tratada de modo a dar suporte à reconstrução dos dados perdidos a partir dos chips de memória restantes. Usando uma análise da IBM e considerando um servidor de 10.000 processadores com 4 GB por processador, gera as seguintes taxas de erros irrecuperáveis em três anos de operação:

- Somente paridade — cerca de 90.000 ou uma falha irrecuperável (ou não detectada) a cada 17 minutos.
- Somente ECC — cerca de 3.500 ou cerca de uma falha não detectada ou irrecuperável a cada 7,5 horas.
- Chipkill — 6 ou cerca de uma falha não detectada ou irrecuperável a cada dois meses.

Outro modo de ver isso é verificar o número máximo de servidores (cada um com 4 GB) que pode ser protegido, ao mesmo tempo que temos a mesma taxa de erros demonstrada para o Chipkill. Para a paridade, mesmo um servidor com um único processador terá uma taxa de erro irrecuperável maior do que um sistema de 10.000 servidores protegido por Chipkill. Para a ECC, um sistema de 17 servidores teria aproximadamente a mesma taxa de falhas que um sistema Chipkill com 10.000 servidores. Portanto, o Chipkill é uma exigência para os servidores de 50.000-100.000 em computadores em escala warehouse (Seção 6.8, no Cap. 6).

2.4 PROTEÇÃO: MEMÓRIA VIRTUAL E MÁQUINAS VIRTUAIS

> Uma máquina virtual é levada a ser uma duplicata eficiente e isolada da máquina real. Explicamos essas noções por meio da ideia de um monitor de máquina virtual (Virtual Machine Monitor — VMM) … um VMM possui três características essenciais: 1) oferece um ambiente para programas que é basicamente idêntico ao da máquina original; 2) os programas executados nesse ambiente mostram, no pior dos casos, apenas pequeno decréscimo na velocidade; 3) está no controle total dos recursos do sistema.
>
> ***Gerald Popek e Robert Goldberg***
>
> "Formal requirements for virtualizable third generation architectures", ***Communications of the ACM*** (julho de 1974).

Segurança e privacidade são dois dos desafios mais irritantes para a tecnologia da informação em 2011. Roubos eletrônicos, geralmente envolvendo listas de números de cartão de crédito, são anunciados regularmente, e acredita-se que muitos outros não sejam relatados. Logo, tanto pesquisadores quanto profissionais estão procurando novas maneiras de tornar os sistemas de computação mais seguros. Embora a proteção de informações não seja limitada ao hardware, em nossa visão segurança e privacidade reais provavelmente envolverão a inovação na arquitetura do computador, além dos sistemas de software.

Esta seção começa com uma revisão do suporte da arquitetura para proteger os processos uns dos outros, por meio da memória virtual. Depois, ela descreve a proteção adicional fornecida a partir das máquinas virtuais, os requisitos de arquitetura das máquinas virtuais e o desempenho de uma máquina virtual. Como veremos no Capítulo 6, as máquinas virtuais são uma tecnologia fundamental para a computação em nuvem.

Proteção via memória virtual

A memória virtual baseada em página, incluindo um TLB (Translation Lookaside Buffer), que coloca em cache as entradas da tabela de página, é o principal mecanismo que protege os processos uns dos outros. As Seções B.4 e B.5, no Apêndice B, revisam a memória virtual, incluindo uma descrição detalhada da proteção via segmentação e paginação no 80x86. Esta subseção atua como uma breve revisão; consulte essas seções se ela for muito rápida.

A multiprogramação, pela qual vários programas executados simultaneamente compartilhariam um computador, levou a demandas por proteção e compartilhamento entre programas e ao conceito de *processo*. Metaforicamente, processo é o ar que um programa respira e seu espaço de vida, ou seja, um programa em execução mais qualquer estado necessário para continuar executando-o. A qualquer instante, deve ser possível passar de um processo para outro. Essa troca é chamada de *troca de processo* ou *troca de contexto*.

O sistema operacional e a arquitetura unem forças para permitir que os processos compartilhem o hardware, sem interferir um com o outro. Para fazer isso, a arquitetura precisa limitar o que um processo pode acessar ao executar um processo do usuário, permitindo ainda que um processo do sistema operacional acesse mais. No mínimo, a arquitetura precisa fazer o seguinte:

1. Oferecer pelo menos dois modos, indicando se o processo em execução é do usuário ou do sistema operacional. Este último processo, às vezes, é chamado de processo *kernel* ou processo *supervisor*.

2. Oferecer uma parte do status do processador que um processo do usuário pode usar, mas não escrever. Esse status inclui bit(s) de modo usuário/supervisor, bit de ativar/desativar exceção e informações de proteção de memória. Os usuários são impedidos de escrever nesse status, pois o sistema operacional não poderá controlar os processos do usuário se eles puderem se dar privilégios de supervisor, desativar exceções ou alterar a proteção da memória.
3. Oferecer mecanismos pelos quais o processador pode ir do modo usuário para o modo supervisor e vice-versa. A primeira direção normalmente é alcançada por uma *chamada do sistema*, implementada como uma instrução especial que transfere o controle para um local determinado no espaço de código do supervisor. O PC é salvo a partir do ponto da chamada do sistema, e o processador é colocado no modo supervisor. O retorno ao modo usuário é como um retorno de sub-rotina, que restaura o modo usuário/supervisor anterior.
4. Oferecer mecanismos para limitar os acessos à memória a fim de proteger o estado da memória de um processo sem ter de passar o processo para o disco em uma troca de contexto.

O Apêndice A descreve vários esquemas de proteção de memória, mas o mais popular é, sem dúvida, a inclusão de restrições de proteção a cada página da memória virtual. As páginas de tamanho fixo, normalmente com 4 KB ou 8 KB de extensão, são mapeadas a partir do espaço de endereços virtuais para o espaço de endereços físicos, por meio de uma tabela de página. As restrições de proteção estão incluídas em cada entrada da tabela de página. As restrições de proteção poderiam determinar se um processo do usuário pode ler essa página, se um processo do usuário pode escrever nessa página e se o código pode ser executado a partir dessa página. Além disso, um processo não poderá ler nem escrever em uma página se não estiver na tabela de página. Como somente o SO pode atualizar a tabela de página, o mecanismo de paginação oferece proteção de acesso total.

A memória virtual paginada significa que cada acesso à memória usa logicamente pelo menos o dobro do tempo, com um acesso à memória para obter o endereço físico e um segundo acesso para obter os dados. Esse custo seria muito alto. A solução é contar com o princípio da localidade, se os acessos tiverem proximidade; então, as *traduções de acesso* para os acessos também precisam ter proximidade. Mantendo essas traduções de endereço em uma cache especial, um acesso à memória raramente requer um segundo acesso para traduzir os dados. Essa cache de tradução de endereço especial é conhecida como Translation Lookaside Buffer (TLB).

A entrada do TLB é como uma entrada de cache em que a tag mantém partes do endereço virtual e a parte de dados mantém um endereço de página físico, campo de proteção, bit de validade e normalmente um bit de utilização e um bit de modificação. O sistema operacional muda esses bits alterando o valor na tabela de página e depois invalidando a entrada de TLB correspondente. Quando a entrada é recarregada da tabela de página, o TLB apanha uma cópia precisa dos bits.

Considerando que o computador obedece fielmente às restrições nas páginas e mapeia os endereços virtuais aos endereços físicos, isso pode parecer o fim. As manchetes de jornal sugerem o contrário.

O motivo pelo qual isso não é o fim é que dependemos da exatidão do sistema operacional e também do hardware. Os sistemas operacionais de hoje consistem em dezenas de milhões de linhas de código. Como os bugs são medidos em número por milhares de linhas de código, existem milhares de bugs nos sistemas operacionais em produção. As falhas no SO levaram a vulnerabilidades que são exploradas rotineiramente.

Esse problema e a possibilidade de que não impor a proteção poderia ser muito mais custoso do que no passado têm levado algumas pessoas a procurarem um modelo de proteção com uma base de código muito menor do que o SO inteiro, como as máquinas virtuais.

Proteção via máquinas virtuais

Uma ideia relacionada à com a memória virtual que é quase tão antiga é a de máquinas virtuais (Virtual Machines — VM). Elas foram desenvolvidas no final da década de 1960 e continuaram sendo uma parte importante da computação por mainframe durante anos. Embora bastante ignoradas no domínio dos computadores monousuários nas décadas de 1980 e 1990, recentemente elas ganharam popularidade devido:

- à crescente importância do isolamento e da segurança nos sistemas modernos;
- às falhas na segurança e confiabilidade dos sistemas operacionais padrão;
- ao compartilhamento de um único computador entre muitos usuários não relacionados;
- aos aumentos drásticos na velocidade bruta dos processadores, tornando o overhead das VMs mais aceitável.

A definição mais ampla das VMs inclui basicamente todos os métodos de emulação que oferecem uma interface de software-padrão, como a Java VM. Estamos interessados nas VMs que oferecem um ambiente completo em nível de sistema, no nível binário da arquitetura do conjunto de instruções (Instruction Set Architecture — ISA). Muitas vezes, a VM suporta a mesma ISA que o hardware nativo. Entretanto, também é possível suportar uma ISA diferente, e tais técnicas muitas vezes são empregadas quando migramos entre ISAs para permitir que o software da ISA original seja usado até que possa ser transferido para a nova ISA. Nosso foco será em VMs onde a ISA apresentada pela VM e o hardware nativo combinam. Essas VMs são chamadas *máquinas virtuais do sistema* (operacional). IBM VM/370, VMware ESX Server e Xen são alguns exemplos. Elas apresentam a ilusão de que os usuários de uma VM possuem um computador inteiro para si mesmos, incluindo uma cópia do sistema operacional. Um único computador executa várias VMs e pode dar suporte a uma série de sistemas operacionais (SOs) diferentes. Em uma plataforma convencional, um único SO "possui" todos os recursos de hardware, mas com uma VM vários SOs compartilham os recursos do hardware.

O software que dá suporte às VMs é chamado *monitor de máquina virtual* (Virtual Machine Monitor — VMM) ou *hipervisor*; o VMM é o coração da tecnologia de máquina virtual. A plataforma de hardware subjacente é chamada de *hospedeiro* (*host*), e seus recursos são compartilhados entre as VMs de *convidadas* (*guests*). O VMM determina como mapear os recursos virtuais aos recursos físicos: um recurso físico pode ser de tempo compartilhado, particionado ou até mesmo simulado no software. O VMM é muito menor do que um SO tradicional; a parte de isolamento de um VMM talvez tenha apenas 10.000 linhas de código.

Em geral, o custo de virtualização do processador depende da carga de trabalho. Os programas voltados a processador em nível de usuário, como o SPEC CPU2006, possuem zero overhead de virtualização, pois o SO raramente é chamado, de modo que tudo é executado nas velocidades nativas. Em geral, cargas de trabalho com uso intenso de E/S também utilizam intensamente o SO, que executa muitas chamadas do sistema e instruções privilegiadas, o que pode resultar em alto overhead de virtualização. O overhead é determinado pelo número de instruções que precisam ser simuladas pelo VMM e pela lentidão com que são emuladas. Portanto, quando as VMs convidadas executam a mesma ISA que o host, conforme presumimos aqui, o objetivo da arquitetura e da VMM é executar quase todas as instruções diretamente no hardware nativo. Por outro lado, se a carga de

trabalho com uso intensivo de E/S também for *voltada para E/S*, o custo de virtualização do processador pode ser completamente ocultado pela baixa utilização do processador, pois ele está constantemente esperando pela E/S.

Embora nosso interesse aqui seja nas VMs para melhorar a proteção, elas oferecem dois outros benefícios que são comercialmente significativos:

1. *Gerenciamento de software*. As VMs oferecem uma abstração que pode executar um conjunto de software completo, incluindo até mesmo sistemas operacionais antigos, como o DOS. Uma implantação típica poderia ser algumas VMs executando SOs legados, muitas executando a versão atual estável do SO e outras testando a próxima versão do SO.
2. *Gerenciamento de hardware.* Um motivo para múltiplos servidores é ter cada aplicação executando com a versão compatível do sistema operacional em computadores separados, pois essa separação pode melhorar a dependência. As VMs permitem que esses conjuntos separados de software sejam executadas independentemente, embora compartilhem o hardware, consolidando assim o número de servidores. Outro exemplo é que alguns VMMs admitem a migração de uma VM em execução para um computador diferente, seja para balancear a carga seja para abandonar o hardware que falha.

É por essas duas razões que os servidores baseados na nuvem, como os da Amazon, contam com máquinas virtuais.

Requisitos de um monitor de máquina virtual

O que um monitor de VM precisa fazer? Ele apresenta uma interface de software para o software convidado, precisa isolar o status dos convidados uns dos outros e proteger-se contra o software convidado (incluindo SOs convidados). Os requisitos qualitativos são:

- O software convidado deve se comportar em uma VM exatamente como se estivesse rodando no hardware nativo, exceto pelo comportamento relacionado com o desempenho ou pelas limitações dos recursos fixos compartilhados por múltiplas VMs.
- O software convidado não deverá ser capaz de mudar a alocação dos recursos reais do sistema diretamente.

Para "virtualizar" o processador, o VMM precisa controlar praticamente tudo — acesso ao estado privilegiado, tradução de endereço, E/S, exceções e interrupções —, embora a VM e o SO convidados em execução os estejam usando temporariamente.

Por exemplo, no caso de uma interrupção de timer, o VMM suspenderia a VM convidada em execução, salvaria seu status, trataria da interrupção, determinaria qual VM convidada será executada em seguida e depois carregaria seu status. As VMs convidadas que contam com interrupção de timer são fornecidas com um timer virtual e uma interrupção de timer simulada pelo VMM.

Para estar no controle, o VMM precisa estar em um nível de privilégio mais alto do que a VM convidada, que geralmente executa no modo usuário; isso também garante que a execução de qualquer instrução privilegiada será tratada pelo VMM. Os requisitos básicos das máquinas virtuais do sistema são quase idênticos àqueles para a memória virtual paginada, que listamos anteriormente.

- Pelo menos dois modos do processador, sistema e usuário.
- Um subconjunto privilegiado de instruções, que está disponível apenas no modo do sistema, resultando em um trap se for executado no modo usuário. Todos os recursos do sistema precisam ser controláveis somente por meio dessas instruções.

(Falta de) Suporte à arquitetura de conjunto de instruções para máquinas virtuais

Se as VMs forem planejadas durante o projeto da ISA, será relativamente fácil reduzir o número de instruções que precisam ser executadas por um VMM e o tempo necessário para simulá-las. Uma arquitetura que permite que a VM execute diretamente no hardware ganha o título de *virtualizável*, e a arquitetura IBM 370 orgulhosamente ostenta este título.

Infelizmente, como as VMs só foram consideradas para aplicações desktop e servidor baseado em PC muito recentemente, a maioria dos conjuntos de instruções foi criada sem que se pensasse na virtualização. Entre esses culpados incluem-se o 80x86 e a maioria das arquiteturas RISC.

Como o VMM precisa garantir que o sistema convidado só interaja com recursos virtuais, um SO convidado convencional é executado como um programa no modo usuário em cima do VMM. Depois, se um SO convidado tentar acessar ou modificar informações relacionadas com os recursos do software por meio de uma instrução privilegiada — por exemplo, lendo ou escrevendo o ponteiro da tabela de página —, ele gerará um trap para o VMM. O VMM pode, então, efetuar as mudanças apropriadas aos recursos reais correspondentes.

Logo, se qualquer instrução tentar ler ou escrever essas informações sensíveis ao trap, quando executada no modo usuário, a VMM poderá interceptá-la e dar suporte a uma versão virtual da informação sensível, como o SO convidado espera.

Na ausência de tal suporte, outras medidas precisam ser tomadas. Um VMM deve tomar precauções especiais para localizar todas as instruções problemáticas e garantir que se comportem corretamente quando executadas por um SO convidado, aumentando assim a complexidade do VMM e reduzindo o desempenho da execução da VM.

As Seções 2.5 e 2.7 oferecem exemplos concretos de instruções problemáticas na arquitetura 80x86.

Impacto das máquinas virtuais sobre a memória virtual e a E/S

Outro desafio é a virtualização da memória virtual, pois cada SO convidado em cada VM gerencia seu próprio conjunto de tabelas de página. Para que isso funcione, o VMM separa as noções de *memória real* e *memória física* (que normalmente são tratadas como sinônimos) e torna a memória real um nível separado, intermediário entre a memória virtual e a memória física (alguns usam os nomes *memória virtual*, *memória física* e *memória de máquina* para indicar os mesmos três níveis). O SO convidado mapeia a memória virtual à memória real por meio de suas tabelas de página, e as tabelas de página do VMM mapeiam a memória real dos convidados à memória física. A arquitetura de memória virtual é especificada por meio de tabelas de página, como no IBM VM/370 e no 80x86, ou por meio da estrutura de TLB, como em muitas arquiteturas RISC.

Em vez de pagar um nível extra de indireção em cada acesso à memória, o VMM mantém uma *tabela de página de sombra* (*shadow page table*), que é mapeada diretamente do espaço de endereço virtual do convidado ao espaço do hardware. Detectando todas as modificações à tabela de página do convidado, o VMM pode garantir que as entradas da tabela de página de sombra sendo usadas pelo hardware para traduções correspondam àquelas do ambiente do SO convidado, com a exceção das páginas físicas corretas substituídas pelas páginas reais nas tabelas convidadas. Logo, o VMM precisa interceptar qualquer tentativa do SO convidado de alterar sua tabela de página ou de acessar o ponteiro da tabela de página. Isso normalmente é feito protegendo a escrita das tabelas de página convidadas e interceptando qualquer acesso ao ponteiro da tabela de página por um SO convidado.

Conforme indicamos, o último acontecerá naturalmente se o acesso ao ponteiro da tabela de página for uma operação privilegiada.

A arquitetura IBM 370 solucionou o problema da tabela de página na década de 1970 com um nível adicional de indireção que é gerenciado pelo VMM. O SO convidado mantém suas tabelas de página como antes, de modo que as páginas de sombra são desnecessárias. A AMD propôs um esquema semelhante para a sua revisão pacífica do 80x86.

Para virtualizar o TLB arquitetado em muitos computadores RISC, o VMM gerencia o TLB real e tem uma cópia do conteúdo do TLB de cada VM convidada. Para liberar isso, quaisquer instruções que acessem o TLB precisam gerar traps. Os TLBs com tags Process ID podem aceitar uma mistura de entradas de diferentes VMs e o VMM, evitando assim o esvaziamento do TBL em uma troca de VM. Nesse meio-tempo, em segundo plano, o VMM admite um mapeamento entre os Process IDs virtuais da VM e os Process IDs reais.

A última parte da arquitetura para virtualizar é a E/S. Essa é a parte mais difícil da virtualização do sistema, devido ao número crescente de dispositivos de E/S conectados ao computador *e* à diversidade crescente de tipos de dispositivo de E/S. Outra dificuldade é o compartilhamento de um dispositivo real entre múltiplas VMs, e outra ainda vem do suporte aos milhares de drivers de dispositivo que são exigidos, especialmente se diferentes OS convidados forem admitidos no mesmo sistema de VM. A ilusão da VM pode ser mantida dando-se a cada VM versões genéricas de cada tipo de driver de dispositivo de E/S e depois deixando para o VMM o tratamento da E/S real.

O método para mapear um dispositivo de E/S virtual para físico depende do tipo de dispositivo. Por exemplo, os discos físicos normalmente são particionados pelo VMM para criar discos virtuais para as VMs convidadas, e o VMM mantém o mapeamento de trilhas e setores virtuais aos equivalentes físicos. As interfaces de rede normalmente são compartilhadas entre as VMs em fatias de tempo muito curtas, e a tarefa do VMM é registrar as mensagens para os endereços de rede virtuais a fim de garantir que as VMs convidadas recebam apenas mensagens enviadas para elas.

Uma VMM de exemplo: a máquina virtual Xen

No início do desenvolvimento das VMs, diversas ineficiências se tornaram aparentes. Por exemplo, um SO convidado gerencia seu mapeamento de página, mas esse mapeamento é ignorado pelo VMM, que realiza o mapeamento real para as páginas físicas. Em outras palavras, quantidade significativa de esforço desperdiçado é gasta apenas para satisfazer o SO convidado. Para reduzir tais ineficiências, os desenvolvedores de VMM decidiram que pode valer a pena permitir que o SO convidado esteja ciente de que está rodando em uma VM. Por exemplo, um SO convidado poderia pressupor uma memória real tão grande quanto sua memória virtual, de modo que nenhum gerenciamento de memória será exigido pelo SO convidado.

A permissão de pequenas modificações no SO convidado para simplificar a virtualização é conhecida como *paravirtualização*, e o VMM Xen, de fonte aberto, é um bom exemplo disso. O VMM Xen oferece a um SO convidado uma abstração de máquina virtual semelhante ao hardware físico, mas sem muitas das partes problemáticas. Por exemplo, para evitar o esvaziamento do TBL, o Xen é mapeado nos 64 MB superiores do espaço de endereços de cada VM. Ele permite que o SO convidado aloque páginas, apenas cuidando para que não infrinja as restrições de proteção. Para proteger o SO convidado contra programas do usuário na VM, o Xen tira proveito dos quatro níveis de proteção disponíveis no 80x86. O VMM Xen é executado no mais alto nível de privilégio (0), o SO

convidado é executado no próximo nível de privilégio (1) e as aplicações são executadas no nível de privilégio mais baixo (3). A maioria dos SOs para o 80x86 mantém tudo nos níveis de privilégio 0 ou 3.

Para que as sub-redes funcionem corretamente, o Xen modifica o SO convidado para não usar partes problemáticas da arquitetura. Por exemplo, a porta do Linux para o Xen alterou cerca de 3.000 linhas, ou cerca de 1% do código específico do 80x86. Porém, essas mudanças não afetam as interfaces binárias da aplicação do SO convidado.

Para simplificar o desafio de E/S das VMs, recentemente o Xen atribuiu máquinas virtuais privilegiadas a cada dispositivo de E/S de hardware. Essas VMs especiais são chamadas *domínios de driver* (o Xen chama suas VMs de "domínios"). Os domínios de driver executam os drivers do dispositivo físico, embora as interrupções ainda sejam tratadas pela VMM antes de serem enviadas para o domínio de driver apropriado. As VMs regulares, chamadas *domínios de convidado*, executam drivers de dispositivo virtuais simples, que precisam se comunicar com os drivers de dispositivo físicos nos domínios de driver sobre um canal para acessar o hardware de E/S físico. Os dados são enviados entre os domínios de convidado e driver pelo remapeamento de página.

2.5 QUESTÕES CRUZADAS: O PROJETO DE HIERARQUIAS DE MEMÓRIA

Esta seção descreve três tópicos abordados em outros capítulos que são fundamentais para as hierarquias de memória.

Proteção e arquitetura de conjunto de instruções

A proteção é um esforço conjunto de arquitetura e sistemas operacionais, mas os arquitetos tiveram de modificar alguns detalhes esquisitos das arquiteturas de conjunto de instruções existentes quando a memória virtual se tornou popular. Por exemplo, para dar suporte à memória virtual no IBM 370, os arquitetos tiveram de alterar a bem-sucedida arquitetura do conjunto de instruções do IBM 360, que havia sido anunciada seis anos antes. Ajustes semelhantes estão sendo feitos hoje para acomodar as máquinas virtuais.

Por exemplo, a instrução POPF do 80x86 carrega os registradores de flag do topo da pilha para a memória. Um dos flags é o flag Interrupt Enable (IE). Se você executar a instrução POPF no modo usuário, em vez de interceptá-la, ela simplesmente mudará todos os flags, exceto IE. No modo do sistema, ela não mudará o IE. Como um SO convidado é executado no modo usuário dentro de uma VM, isso é um problema, pois ele espera ver o IE alterado. Extensões da arquitetura 80x86 para suportar a virtualização eliminaram esse problema.

Historicamente, o hardware de mainframe IBM e o VMM utilizavam três passos para melhorar o desempenho das máquinas virtuais:

1. Reduzir o custo da virtualização do processador.
2. Reduzir o custo de overhead de interrupção devido à virtualização.
3. Reduzir o custo de interrupção direcionando as interrupções para a VM apropriada sem invocar o VMM.

A IBM ainda é o padrão dourado da tecnologia de máquina virtual. Por exemplo, um mainframe IBM executava milhares de VMs Linux em 2000, enquanto o Xen executava 25 VMs, em 2004 (Clark *et al.*, 2004). Versões recentes de chipsets adicionaram instruções especiais para suportar dispositivos em uma VM, para mascarar interrupções em níveis inferiores de cada VM e para direcionar interrupções para a VM apropriada.

Consistência dos dados em cache

Os dados podem ser encontrados na memória e na cache. Desde que um processador seja o único dispositivo a alterar ou ler os dados e a cache fique entre o processador e a memória, haverá pouco perigo de o processador ver a cópia antiga ou *desatualizada* (*stale*). Conforme mencionaremos no Capítulo 4, múltiplos processadores e dispositivos de E/S aumentam a oportunidade de as cópias serem inconsistentes e de lerem a cópia errada.

A frequência do problema de coerência de cache é diferente para multiprocessadores e para E/S. Múltiplas cópias de dados são um evento raro para E/S — que deve ser evitado sempre que possível —, mas um programa em execução em múltiplos processadores *desejará* ter cópias dos mesmos dados em várias caches. O desempenho de um programa multiprocessador depende do desempenho do sistema ao compartilhar dados.

A questão de *coerência da cache de E/S* é esta: onde ocorre a E/S no computador — entre o dispositivo de E/S e a cache ou entre o dispositivo de E/S e a memória principal? Se a entrada colocar dados na cache e a saída ler dados da cache, tanto a E/S quanto o processador verão os mesmos dados. A dificuldade dessa técnica é que ela interfere com o processador e pode fazer com que o processador pare para a E/S. A entrada também pode interferir com a cache, deslocando alguma informação com dados novos que provavelmente serão acessados em breve.

O objetivo para o sistema de E/S em um computador com cache é impedir o problema dos dados desatualizados, enquanto interfere o mínimo possível. Muitos sistemas, portanto, preferem que a E/S ocorra diretamente na memória principal, com a memória principal atuando como um buffer de E/S. Se uma cache write-through for usada, a memória terá uma cópia atualizada da informação e não haverá o problema de dados passados para a saída (esse benefício é um motivo para os processadores usarem a cache write-through). Infelizmente, hoje o write-through normalmente é encontrado apenas nas caches de dados de primeiro nível, apoiados por uma cache L2 que use write-back.

A entrada requer algum trabalho extra. A solução de software é garantir que nenhum bloco do buffer de entrada esteja na cache. Uma página contendo o buffer pode ser marcada como não passível de cache (noncachable), e o sistema operacional sempre poderá entrar em tal página. Como alternativa, o sistema operacional pode esvaziar os endereços de buffer da cache antes que ocorra a entrada. Uma solução de hardware é verificar os endereços de E/S na entrada para ver se eles estão na cache. Se houver uma correspondência de endereços de E/S na cache, as entradas de cache serão invalidadas para evitar dados passados. Todas essas técnicas também podem ser usadas para a saída com caches write-back.

A consistência da cache do processador é um assunto essencial na era dos processadores multicore, e vamos examiná-la em detalhes no Capítulo 5.

2.6 JUNTANDO TUDO: HIERARQUIA DE MEMÓRIA NO ARM CORTEX-A8 E INTEL CORE I7

Esta seção desvenda as hierarquias de memória do ARM Cortex-A8 (daqui em diante chamado Cortex-A8) e do Intel Core i7 (daqui em diante chamado i7) e mostra o desempenho de seus componentes para um conjunto de benchmarks de thread único. Nós examinamos o Cortex-A8 primeiro porque ele tem um sistema de memória mais simples. Vamos entrar em mais detalhes sobre o i7 detalhando uma referência de memória. Esta seção supõe que os leitores estejam familiarizados com a organização de uma hierarquia de cache de dois níveis usando caches indexadas virtualmente. Os elementos básicos de tal sistema de memória são explicados em detalhes no Apêndice B, e os leitores que não estão acos-

tumados com a organização desses sistemas são enfaticamente aconselhados a revisar o exemplo do Opteron no Apêndice B. Após a compreensão da organização do Opteron, a breve explicação sobre o sistema Cortex-A8, que é similar, será fácil de acompanhar.

O ARM Cortex-A8

O Cortex-A8 é um núcleo configurável que dá suporte à arquitetura de conjunto de instruções ARMv7. Ele é fornecido como um núcleo IP (propriedade intelectual). Os núcleos IP são a forma dominante de entrega de tecnologia nos mercados dos embarcados, PMD e relacionados. Bilhões de processadores ARM e MIPS foram criados a partir desses núcleos IP. Observe que eles são diferentes dos núcleos no Intel i7 ou AMD Athlon multicores. Um núcleo IP (que pode ser, ele próprio, um multicore) é projetado para ser incorporado com outras lógicas (uma vez que ele é o núcleo de um chip), incluindo processadores de aplicação específica (como um codificador ou um decodificador de vídeo), interfaces de E/S e interfaces de memória, e então fabricados para gerar um processador otimizado para uma aplicação em particular. Por exemplo, o núcleo Cortex-A8 IP é usado no Apple iPad e smartphones de diversos fabricantes, incluindo Motorola e Samsung. Embora o núcleo do processador seja quase idêntico, os chips resultantes têm muitas diferenças.

Geralmente, os núcleos IP têm dois tipos: núcleos hard são otimizados para um fornecedor particular de semicondutores e são caixas-pretas com interfaces externas (mas ainda no chip). Em geral, permitem a parametrização somente da lógica fora do núcleo, como tamanhos de cache L2, sendo que o núcleo IP não pode ser modificado. Normalmente núcleos soft são fornecidos em uma forma que usa uma biblioteca-padrão de elementos lógicos. Um núcleo soft pode ser compilado para diferentes fornecedores de semicondutores e também pode ser modificado, embora modificações extensas sejam difíceis, devido à complexidade dos núcleos IP modernos. Em geral, núcleos hard apresentam melhor desempenho e menor área de substrato, enquanto os núcleos soft permitem o atendimento a outros fornecedores e podem ser modificados mais facilmente.

O Cortex-A8 pode enviar duas instruções por clock a taxas de clock de até 1 GHz. Ele pode suportar uma hierarquia de cache de dois níveis, com o primeiro nível sendo um par de caches (para I & D), cada uma com 16 KB ou 32 KB organizados como associativos por conjuntos de quatro vias e usando previsão de via e susbstituição aleatória. O objetivo é ter latência de acesso de ciclo único para as caches, permitindo que o Cortex-A8 mantenha um atraso de carregamento para uso de um ciclo, busca de instruções mais simples e menor penalidade por busca de instrução correta quando uma falta de desvio faz com que a instrução errada seja lida na pré-busca. A cache de segundo nível opcional, quando presente, é um conjunto associativo de oito vias e pode ser configurado com 128 KB até 1 MB. Ela é organizada em 1-4 bancos para permitir que várias transferências de memória ocorram ao mesmo tempo. Um barramento externo de 64-128 bits trata as requisições de memória. A cache de primeiro nível é indexada virtualmente e *taggeada* fisicamente, e a cache de segundo nível é indexada e *taggeada* fisicamente. Os dois níveis usam um tamanho de bloco de 64 bytes. Para a D-cache de 32 KB e um tamanho de página de 4 KB, cada página física pode mapear dois endereços de cache diferentes. Tais instâncias são evitadas por detecção de hardware em uma falta, como na Seção B.3 do Apêndice B.

O gerenciamento de memória é feito por um par de TLBs (I e D), cada um dos quais é totalmente associativo com 32 entradas e tamanho de página variável (4 KB, 16 KB, 64 KB, 1 MB e 16 MB). A substituição no TLB é feita por um algoritmo *round robin*. As faltas do TLB são tratadas no hardware, que percorre uma estrutura de tabela de página na memória. A Figura 2.16 mostra como o endereço virtual de 32 bits é usado para indexar

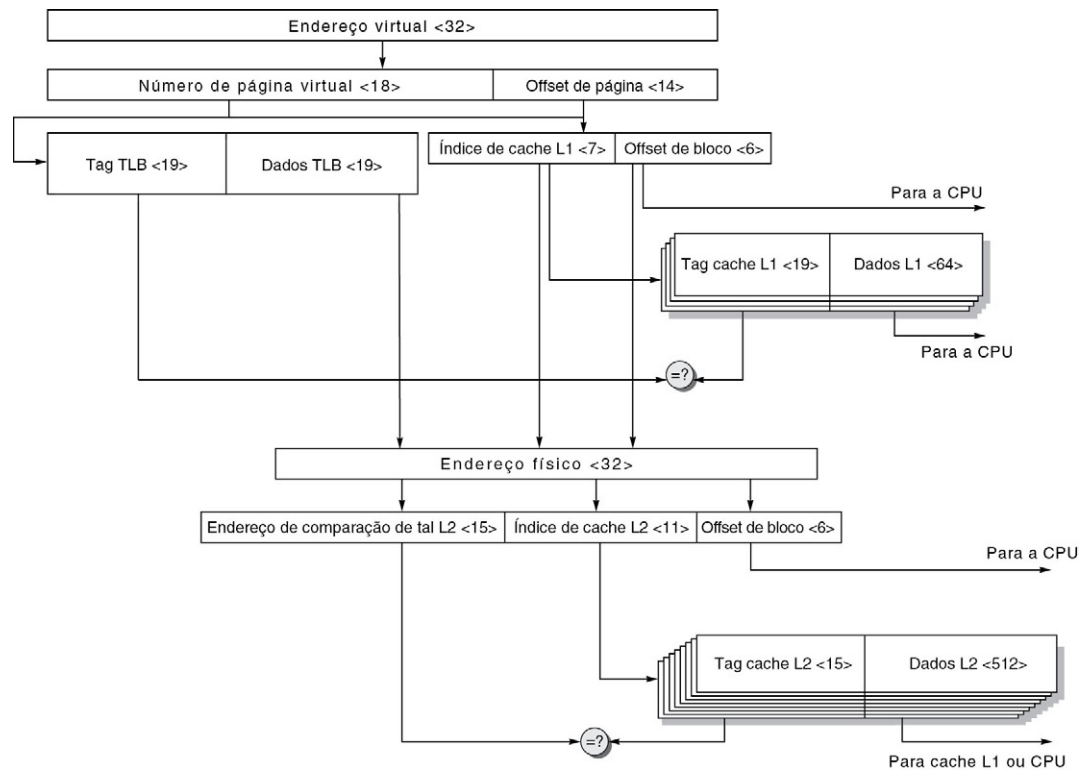



FIGURA 2.16 **Endereço virtual, endereço físico, índices, tags e blocos de dados para as caches de dados e TLB de dados do ARM Cortex A-8.**
Uma vez que as hierarquias de instrução e dados são simétricas, mostramos somente uma. A TLB (instrução ou dados) é totalmente associativa com 32 entradas. A cache L1 é associativa por conjunto de quatro vias com blocos de 64 bytes e capacidade de 32 KB. A cache L2 é associativa por conjunto com oito vias com blocos de 64 bytes e capacidade de 1 MB. Esta figura não mostra os bits de validade e bits de proteção para as caches e TLB nem o uso de bits de modo de predição que ditam o banco de predição da cache L1.

a TLB e as caches, supondo caches primárias de 32 KB e uma cache secundária de 512 KB com tamanho de página de 16 KB.

Desempenho da hierarquia de memória do ARM Cortex-A8

A hierarquia de memória do Cortex-A8 foi simulada com caches primárias de 32 KB e uma cache L2 associativa por conjunto de oito vias de 1 MB, usando os benchmarks inteiros Minnespec (**KleinOswski e Lilja, 2002**). O Minnespec é um conjunto de benchmarks que consiste nos benchmarks SPEC2000, porém com entradas diferentes que reduzem os tempos de execução em várias ordens de magnitude. Embora o uso de entradas menores não mude o *mix* de instruções, ele afeta o comportamento da cache. Por exemplo, em mcf, o benchmark inteiro mais pesado em termos de memória do SPEC2000, o Minnespec, tem taxa de falta, para uma cache de 32 KB, de somente 65% da taxa de falta para a versão SPEC completa. Para uma cache de 1 MB, a diferença é um fator de 6! Em muitos outros benchmarks, as taxas são similares àquelas do mcf, mas as taxas de falta absolutas são muito menores. Por essa razão, não é possível comparar os benchmarks Minnespec com os benchmarks SPEC2000. Em vez disso, os dados são úteis para a análise do impacto relativo às faltas em L1 e L2 e na CPI geral, como faremos no próximo capítulo.

As taxas de falta da cache de instrução para esses benchmarks (e também para as versões completas do SPEC2000, nas quais o Minnespec se baseia) são muito pequenas, mesmo

para o L1: perto de zero para a maioria e abaixo de 1% para todos eles. Essa taxa baixa provavelmente resulta da natureza computacionalmente intensa dos programas SPEC e da cache associativa por conjunto de quatro vias, que elimina a maioria das faltas por conflito. A Figura 2.17 mostra os resultados da cache de dados, que tem taxas de falta significativas para L1 e L2. A penalidade de falta do L1 para um Cortex-A8 de 1 GHz é de 11 ciclos de clock, enquanto a penalidade de falta do L2 é de 60 ciclos de clock, usando SDRAMs DDR como memória principal. Usando essas penalidades de falta, a Figura 2.18 mostra a penalidade média por acesso aos dados. No Capítulo 3, vamos examinar o impacto das faltas de cache na CPI geral.

O Intel Core i7

O i7 suporta a arquitetura de conjunto de instruções x86-64, uma extensão de 64 bits da arquitetura 80x86. O i7 é um processador de execução fora de ordem que inclui quatro núcleos. Neste capítulo, nos concentramos no projeto do sistema de memória e desempenho do ponto de vista de um único núcleo. O desempenho do sistema dos projetos de multiprocessador, incluindo o i7 multicore, será examinado em detalhes no Capítulo 5.

Cada núcleo em um i7 pode executar até quatro instruções 80x86 por ciclo de clock, usando um pipeline de 16 estágios, dinamicamente escalonados, que descreveremos em detalhes no Capítulo 3. O i7 pode também suportar até dois threads simultâneos por processador, usando uma técnica chamada multithreading simultâneo, que será descrita

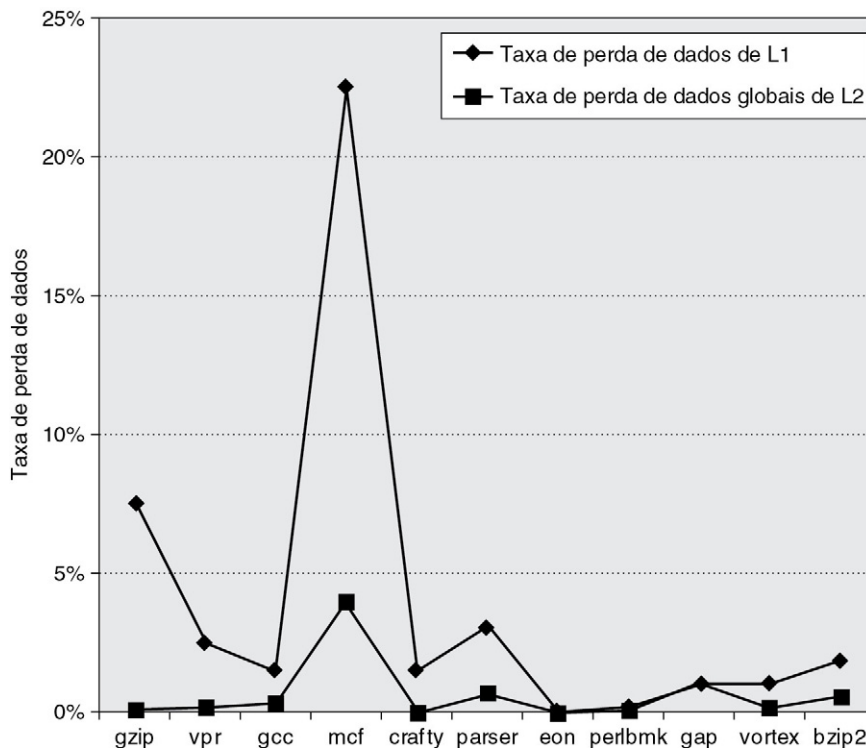



FIGURA 2.17 A taxa de falta de dados para o ARM com uma L1 de 32 KB e a taxa de falta de dados globais de uma L2 de 1 MB usando os benchmarks inteiros do Minnespec é afetada significativamente pelas aplicações.

As aplicações com necessidades de memória maiores tendem a ter taxas de falta maiores, tanto em L1 quanto em L2. Note que a taxa de L2 é a taxa de falta global, que considera todas as referências, incluindo aquelas que acertam em L1. O Mcf é conhecido como cache *buster*.

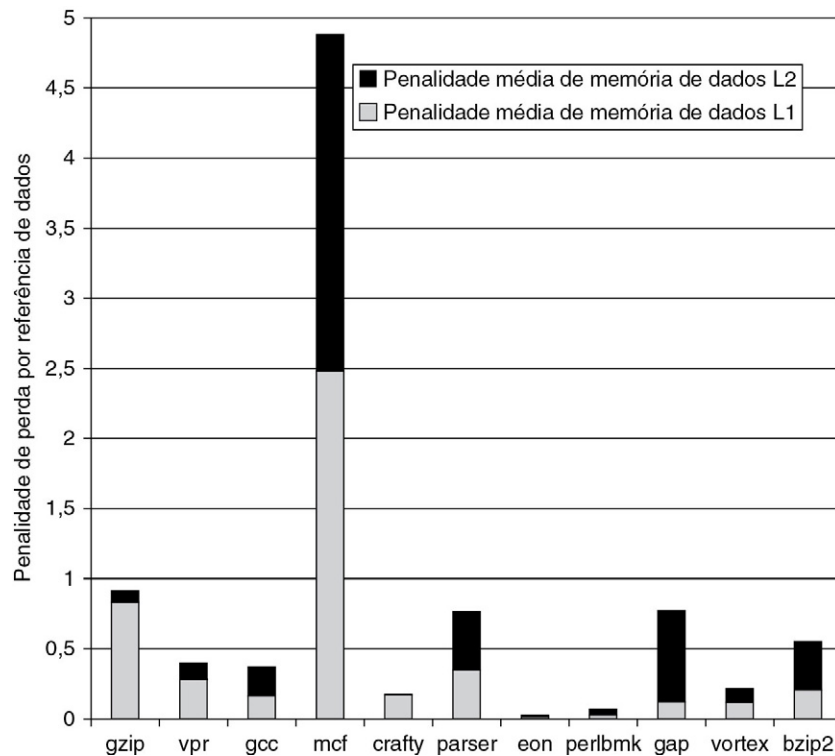



FIGURA 2.18 A penalidade média de acesso à memória por referência da memória de dados vindo de L1 e L2 é mostrada para o processador ARM executando o Minnespec.
Embora as taxas de falta para L1 sejam significativamente maiores, a penalidade de falta de L2, que é mais de cinco vezes maior, significa que as faltas de L2 podem contribuir significativamente.

no Capítulo 4. Em 2010, o i7 mais rápido tinha taxa de clock de 3,3 GHz, que gera um pico de taxa de execução de instruções de 13,2 bilhões de instruções por segundo, ou mais de 50 bilhões de instruções por segundo para o projeto de quatro núcleos.

O i7 pode suportar até três canais de memória, cada qual consistindo em um conjunto de DIMMs separados, e cada um dos quais pode transferir em paralelo. Usando DDR3-1066 (DIMM PC8500), o i7 tem pico de largura de banda de memória pouco acima de 25 GB/s.

O i7 usa endereços virtuais de 48 bits e endereços físicos de 36 bits, gerando uma memória física máxima de 36 GB. O gerenciamento de memória é tratado com um TLB de dois níveis (Apêndice B, Seção B.3), resumido na Figura 2.19.

A Figura 2.20 resume a hierarquia de cache em três níveis do i7. As caches de primeiro nível são indexadas virtualmente e *taggeadas* fisicamente (Apêndice B, Seção B.3), enquanto as caches L2 e L3 são indexadas fisicamente. A Figura 2.21 mostra os passos de um acesso à hierarquia de memória. Primeiro, o PC é enviado para a cache de instruções. O índice da cache de instruções é

$$2^{\text{Índice}} = \frac{\text{Tamanho da cache}}{\text{Tamanho do bloco} \times \text{Associabilidade do conjunto}} = \frac{32\text{K}}{64 \times 4} = 128 = 2^7$$

ou 7 bits. A estrutura de página do endereço da instrução (36 = 48 − 12 bits) é enviada para o TLB de instrução (passo 1). Ao mesmo tempo, o índice de 7 bits (mais 2 bits adicionais para o offset do bloco para selecionar os 16 bytes apropriados, a quantidade de busca de instrução) do endereço virtual é enviado para a cache de instrução (passo 2). Observe

Característica	TLB de instrução	DLB de dados	TLB de segundo nível
Tamanho	128	64	512
Associatividade	4 vias	4 vias	4 vias
Substituição	Pseudo-LRU	Pseudo-LRU	Pseudo-LRU
Latência de acesso	1 ciclo	1 ciclo	6 ciclos
Falta	7 ciclos	7 ciclos	Centenas de ciclos para acessar a tabela de página

FIGURA 2.19 Características da estrutura de TLB do i7, que tem TLBs de primeiro nível de instruções e dados separadas, as duas suportadas, em conjunto, por um TLB de segundo nível.
Os TLBs de primeiro nível suportam o tamanho-padrão de página de 4 KB, além de ter número limitado de entradas de páginas grandes de 2-4 MB. Somente as páginas de 4 KB são suportadas no TLB de segundo nível.

Característica	L1	L2	L3
Tamanho	32 KB I/32 KB D	256 KB	2 MB por núcleo
Associatividade	4 vias I/8 vias D	8 vias	16 vias
Latência de acesso	4 ciclos, com pipeline	10 ciclos	35 ciclos
Esquema de substituição	Pseudo-LRU	Pseudo-LRU	Pseudo-LRU, mas com um algoritmo de seleção ordenada

FIGURA 2.20 Características da hierarquia de cache em três níveis no i7.
Os três caches usam write-back e tamanho de bloco de 64 bytes. As caches L1 e L2 são separadas para cada núcleo, enquanto a cache L3 é compartilhada entre os núcleos em um chip e tem um total de 2 MB por núcleo. As três caches não possuem bloqueio e permitem múltiplas escritas pendentes. Um write buffer merge é usado para a cache L1, que contém dados no evento de que a linha não está presente em L1 quando ela é escrita (ou seja, a falta de escrita em L1 não faz com que a linha seja alocada). L3 é inclusivo de L1 e L2; exploramos essa propriedade em detalhes quando explicamos as caches multiprocessador. A substituição é por uma variante na pseudo-LRU: no caso de L3, o bloco substituído é sempre a via de menor número cujo bit de acesso esteja desligado. Isso não é exatamente aleatório, mas é fácil de computar.

que, para a cache de instrução associativa de quatro vias, 13 bits são necessários para o endereço de cache: 7 bits para indexar a cache, mais 6 bits de offset de bloco para bloco de 64 bytes, mas o tamanho da página é de 4 KB = 2^{12}, o que significa que 1 bit do índice de cache deve vir do endereço virtual. Esse uso de 1 bit de endereço virtual significa que o bloco correspondente poderia, na verdade, estar em dois lugares diferentes da cache, uma vez que o endereço físico correspondente poderia ser um 0 ou 1 nesse local. Para instruções, isso não é um problema, uma vez que, mesmo que uma instrução apareça na cache em dois locais diferentes, as duas versões devem ser iguais. Se tal duplicação de dados, ou *aliasing*, for permitida, a cache deverá ser verificada quando o mapa da página for modificado, o que é um evento pouco frequente. Observe que um uso muito simples da colorização de página (Apêndice B, Seção B.3) pode eliminar a possibilidade desses aliases. Se páginas virtuais de endereço par forem mapeadas para páginas físicas de endereço par (e o mesmo ocorrer com as páginas ímpares), esses aliases poderão não ocorrer, porque os bits de baixa ordem no número das páginas virtual e física serão idênticos.

A TBL de instrução é acessada para encontrar uma correspondência entre o endereço e uma entrada de tabela de página (Page Table Entry — PTE) válida (passos 3 e 4). Além de traduzir o endereço, a TBL verifica se a PTE exige que esse acesso resulte em uma exceção, devido a uma violação de acesso.

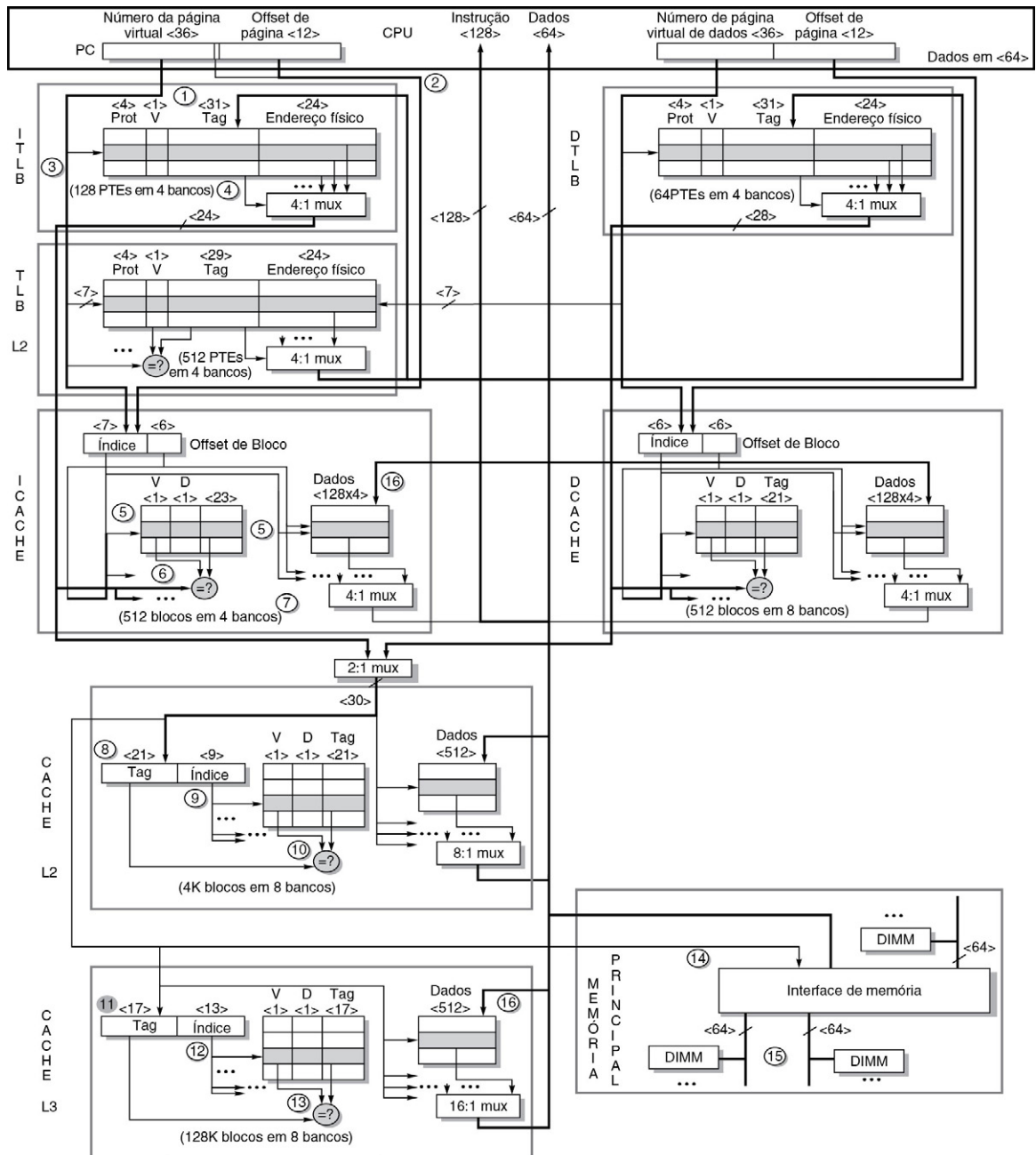



FIGURA 2.21 A hierarquia de memória do Intel i7 e os passos no acesso às instruções e aos dados.
Mostramos somente as leituras de dados. As escritas são similares, no sentido de que começam com uma leitura (uma vez que as caches são write-back). Faltas são tratadas simplesmente colocando os dados em um buffer de escrita, uma vez que a cache L1 não é alocado para escrita.

Uma falta de TLB de instrução primeiro vai para a TLB L2, que contém 512 PTEs com tamanho de página de 4 KB, e é associativa por conjunto com quatro vias. Ela leva dois ciclos de clock para carregar a TLB L1 da TLB L2. Se a TLB L2 falhar, um algoritmo de hardware será usado para percorrer a tabela da página e atualizar a entrada da TLB. No pior caso, a página não estará na memória, e o sistema operacional recuperará a página do disco. Uma vez que milhões de instruções podem ser executadas durante uma falha

de página, o sistema operacional vai realizar outro processo se um estiver esperando para ser executado. Se não houver exceção de TLB, o acesso à cache de instrução continuará.

O campo de índice do endereço é enviado para os quatro bancos da cache de instrução (passo 5). A tag da cache de instrução tem 36 − 7 bits (índice) – 6 bits (offset de bloco), ou 23 bits. As quatro tags e os bits válidos são comparados à estrutura física da página a partir da TLB de instrução (passo 6). Como o i7 espera 16 bytes a cada busca de instrução, 2 bits adicionais são usados do offset de bloco de 6 bits para selecionar os 16 bytes apropriados. Portanto, 7 + 2 ou 9 bits são usados para enviar 16 bytes de instruções para o processador. A cache L1 é pipelining e a latência de um acerto é de quatro ciclos de clock (passo 7). Uma falta vai para a cache de segundo nível.

Como mencionado, a cache de instrução é virtualmente endereçada e fisicamente taggeada. Uma vez que as caches de segundo nível são endereçadas fisicamente, o endereço físico da página da TLB é composta com o offset de página para criar um endereço para acessar a cache L2. O índice L2 é

$$2^{\text{Índice}} = \frac{\text{Tamanho da cache}}{\text{Tamanho do bloco} \times \text{Associabilidade do conjunto}} = \frac{256\,\text{K}}{64 \times 8} = 512 = 2^9$$

então o endereço de bloco de 30 bits (endereço físico de 36 bits – offset de bloco de 6 bits) é dividido em uma tag de 21 bits e um índice 9 bits (passo 8). Uma vez mais, o índice e a tag são enviados para os oitos bancos da cache L2 unificada (passo 9), que são comparados em paralelo. Se um corresponder e for válido (passo 10), é retornado ao bloco em ordem sequencial após a latência inicial de 10 ciclos a uma taxa de 8 bytes por ciclo de clock.

Se a cache L2 falhar, a cache L3 será acessada. Para um i7 de quatro núcleos, que tem uma L3 de 8 MB, o tamanho do índice é

$$2^{\text{Índice}} = \frac{\text{Tamanho da cache}}{\text{Tamanho do bloco} \times \text{Associabilidade do conjunto}} = \frac{8\,\text{M}}{64 \times 16} = 8.192 = 2^{13}$$

O índice de 13 bits (passo 11) é enviado para os 16 bancos de L3 (passo 12). A tag L3, que tem 36 – (13 + 6) = 17 bits, é comparada com o endereço físico da TLB (passo 13). Se ocorrer um acerto, o bloco é retornado depois de uma latência inicial a uma taxa de 16 bytes por clock e colocado em L1 e L3. Se L3 falhar, um acesso de memória será iniciado.

Se a instrução não for encontrada na cache L3, o controlador de memória do chip deverá obter o bloco da memória principal. O i7 tem três canais de memória de 64 bits, que podem agir como um canal de 192 bits, uma vez que existe somente um controlador de memória e o mesmo endereço é enviado nos dois canais (passo 14). Transferências amplas ocorrem quando os dois canais têm DIMMs idênticos. Cada canal pode suportar até quatro DIMMs DDR (passo 15). Quando os dados retornam, são posicionados em L3 e L1 (passo 16), pois L3 é inclusiva.

A latência total da falta de instrução atendida pela memória principal é de aproximadamente 35 ciclos de processador para determinar que uma falta de L3 ocorreu, mais a latência da DRAM para as instruções críticas. Para uma SDRAM DDR1600 de banco único e uma CPU de 3,3 GHz, a latência da DRAM é de cerca de 35 ns ou 100 ciclos de clock para os primeiros 16 bytes, levando a uma penalidade de falta total de 135 ciclos de clock. O controlador de memória preenche o restante do bloco de cache de 64 bytes a uma taxa de 16 bits por ciclo de clock de memória, o que leva mais 15 ns ou 45 ciclos de clock.

Uma vez que a cache de segundo nível é uma cache write-back, qualquer falta pode levar à reescrita de um bloco velho na memória. O i7 tem um write buffer merge de 10 entradas que escreve linhas modificadas de cache quando o próximo nível da cache não é usado

para uma leitura. O buffer de escrita é pesquisado em busca de qualquer falta para ver ser a linha de cache existe no buffer; em caso positivo, a falta é preenchida a partir do buffer. Um buffer similar é usado entre as caches L1 e L2.

Se essa instrução inicial for um *load*, o endereço de dados será enviado para a cache de dados e TLBs de dados, agindo de modo muito similar a um acesso de cache de instrução com uma diferença-chave. A cache de dados de primeiro nível é um conjunto associativo de oito vias, o que significa que o índice é de 6 bits (contra 7 da cache de instrução) e o endereço usado para acessar a cache é o mesmo do offset de página. Portanto, aliases na cache de dados não são um problema.

Suponha que a instrução seja um *store* em vez de um *load*. Quando o store é iniciado, ele realiza uma busca na cache de dados, assim como um load. Uma falta faz com que o bloco seja posicionado em um buffer de escrita, uma vez que a cache L1 não aloca o bloco em uma falta de escrita. Em um acerto, o store não atualiza a cache L1 (ou L2) até mais tarde, depois que se sabe que ele é não especulativo. Durante esse tempo, o store reside em uma fila load-store, parte do mecanismo de controle fora de ordem do processador.

O i7 também suporta pré-busca para L1 e L2 do próximo nível na hierarquia. Na maioria dos casos, a linha pré-obtida é simplesmente o próximo bloco da cache. Ao executar a pré-busca somente para L1 e L2, são evitadas as buscas caras e desnecessárias na memória.

Desempenho do sistema de memória do i7

Nós avaliamos o desempenho da estrutura de cache do i7 usando 19 dos benchmarks SPEC CPU2006 (12 inteiros e sete de ponto flutuante), que foram descritos no Capítulo 1. Os dados desta seção foram coletados pelo professor Lu Peng e pelo doutorando Ying Zhang, ambos da Universidade do Estado da Louisiana.

Começamos com a cache L1. A cache de instrução associativa por conjunto com quatro vias leva a uma taxa de falta de instrução muito baixa, especialmente porque a pré-busca de instrução no i7 é bastante efetiva. Obviamente, avaliar a taxa de falta é um pouco complicado, já que o i7 não gera requisições individuais para unidades de instrução únicas, mas, em vez disso, pré-busca 16 bytes de dados de instrução (em geral, 4-5 instruções). Se, por simplicidade, examinarmos a taxa de falta da cache de instrução como tratamos as referências de instrução únicas, a taxa de falta de cache de instrução do L1 variará entre 0,1-1,8%, com uma média pouco acima de 0,4%. Essa taxa está de acordo com outros estudos do comportamento da cache de instrução para os benchmarks SPEC CPU2006, que mostraram baixas taxas de falta da cache de instrução.

A cache de dados L1 é mais interessante e também a mais complicada de avaliar por três razões:

1. Como a cache de dados L1 não é alocada para escrita, as escritas podem acertar mas nunca errar de verdade, no sentido de que uma escrita que não acerta simplesmente coloca seus dados no buffer de escrita e não registra uma falha.
2. Como, às vezes, a especulação pode estar errada (veja discussão detalhada no Cap. 3), existem referências à cache de dados L1 que não correspondem a loads ou stores que eventualmente completam a execução. Como tais faltas deveriam ser tratadas?
3. Por fim, a cache de dados L1 realiza pré-busca automática. As pré-buscas que falham deveriam ser contadas? Caso afirmativo, como?

Para tratar desses problemas e ao mesmo tempo manter uma quantidade de dados razoável, a Figura 2.22 mostra as faltas de cache de dados L1 de dois modos: 1) relativas ao número de loads que realmente são completados (muitas vezes chamados *graduação* ou *aposentadoria*) e 2) relativas a todos os acessos a cache de dados L1 por qualquer fonte.

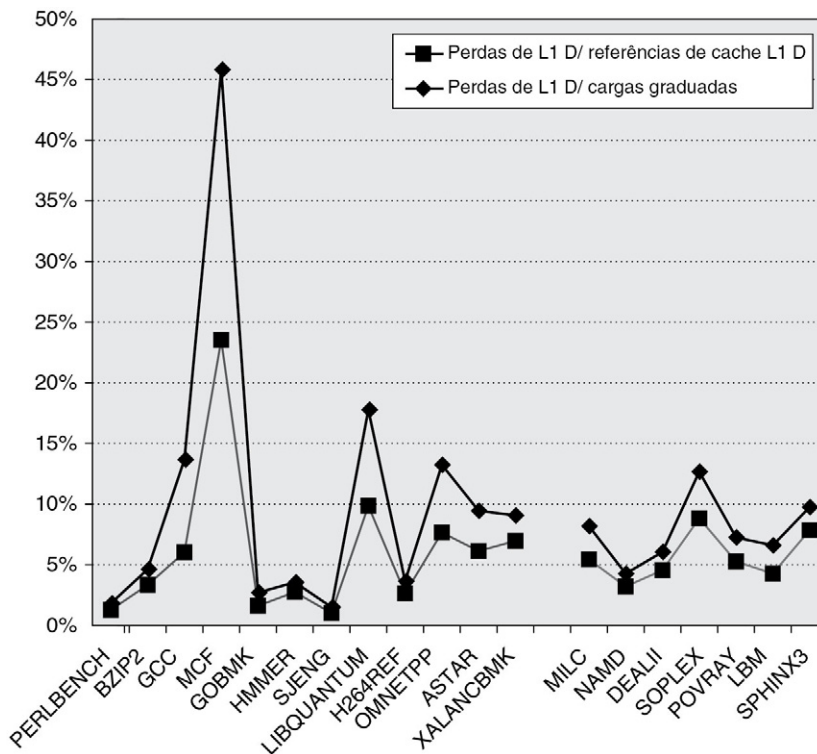



FIGURA 2.22 **A taxa de falta da cache de dados L1 para 17 benchmarks SPEC CPU2006 é mostrada de dois modos: relativa às cargas reais que completam com sucesso a execução e relativa a todas as referências a L1, que também inclui pré-buscas, cargas especulativas que não são completadas, e escritas, que contam como referências, mas não geram faltas.**

Esses dados, como o resto desta seção, foram coletados pelo professor Lu Peng e pelo doutorando Ying Zhang, ambos da Universidade do Estado da Louisiana, com base em estudos anteriores do Intel Core Duo e outros processadores (Peng *et al.*, 2008).

Como veremos, a taxa de faltas, quando medida em comparação somente com os loads completos, é 1,6 vez maior (uma média de 9,5% contra 5,9%). A Figura 2.23 mostra os mesmos dados em forma de tabela.

Com as taxas de falta da cache de dados L1 sendo de 5-10%, e às vezes mais alta, a importância das caches L2 e L3 deve ser óbvia. A Figura 2.24 mostra as taxas de falta das caches L2 e L3 contra o número de referências de L1 (e a Fig. 2.25 mostra os dados em forma de tabela). Uma vez que o custo de uma falta para a memória é de mais de 100 ciclos e a taxa média de falta de dados em L2 é de 4%, L3 é obviamente crítico. Sem L3 e supondo que cerca de metade das instruções é de loads ou stores, as faltas da cache L2 poderiam adicionar dois ciclos por instrução para a CPI! Em comparação, a taxa de falta de dados em L3, de 1%, ainda é significativa, mas quatro vezes menor do que a taxa de falta de L2 e seis vezes menor do que a taxa de falta de L1. No Capítulo 3, vamos examinar o relacionamento entre a CPI do i7 e as faltas de cache, assim como outros efeitos de pipeline.

2.7 FALÁCIAS E ARMADILHAS

Como a mais naturalmente quantitativa das disciplinas da arquitetura de computador, a hierarquia de memória poderia parecer menos vulnerável a falácias e armadilhas. Entretanto, fomos limitados aqui não pela falta de advertências, mas pela falta de espaço!

Benchmark	Faltas de dados L1/loads completos	Faltas de dados L1/referência à cache de dados L1
PERLBENCH	2%	1%
BZIP2	5%	3%
GCC	14%	6%
MCF	46%	24%
GOBMK	3%	2%
HMMER	4%	3%
SJENG	2%	1%
LIBQUANTUM	18%	10%
H264REF	4%	3%
OMNETPP	13%	8%
ASTAR	9%	6%
XALANCBMK	9%	7%
MILC	8%	5%
NAMD	4%	3%
DEALII	6%	5%
SOPLEX	13%	9%
POVRAY	7%	5%
LBM	7%	4%
SPHINX3	10%	8%

FIGURA 2.23 As faltas da cache de dados primários são mostradas em comparação com todos os carregamentos que são completados e todas as referências (que incluem requisições especulativas e pré-buscas).

Falácia. *Prever o desempenho da cache de um programa a partir de outro.*

A Figura 2.26 mostra as taxas de falta de instrução e as taxas de falta de dados para três programas do pacote de benchmark SPEC2000 à medida que o tamanho da cache varia. Dependendo do programa, as faltas de dados por mil instruções para uma cache de 4.096 KB é de 9, 2 ou 90, e as faltas de instrução por mil instruções para uma cache de 4 KB é de 55, 19 ou 0,0004. Programas comerciais, como os bancos de dados, terão taxas de falta significativas até mesmo em grandes caches de segundo nível, o que geralmente não é o caso para os programas SPEC. Claramente, generalizar o desempenho da cache de um programa para outro não é sensato. Como a Figura 2.24 nos lembra, há muita variação, e as previsões sobre as taxas de falta relativas de programas pesados em inteiros e ponto flutuante podem estar erradas, como o mcf eo sphinx3 nos lembram!

Armadilha. *Simular instruções suficientes para obter medidas de desempenho precisas da hierarquia de memória.*

Existem realmente três armadilhas aqui. Uma é tentar prever o desempenho de uma cache grande usando um rastreio pequeno. Outra é que o comportamento da localidade de um programa não é constante durante a execução do programa inteiro. A terceira é que o comportamento da localidade de um programa pode variar de acordo com a entrada.

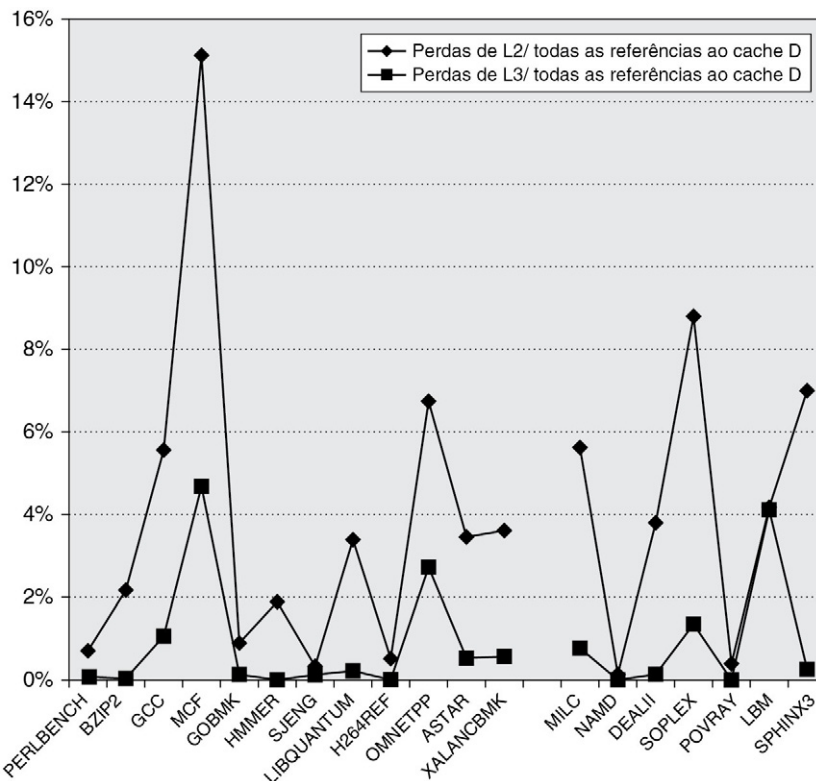



FIGURA 2.24 As taxas de falta das caches de dados L2 e L3 para 17 benchmarks SPEC CPU2006 são mostradas em relação às referências a L1, que também incluem pré-buscas, carregamentos especulativos que não são completados e carregamentos e armazenamentos gerados por programa.
Esses dados, como o resto desta seção, foram coletados pelo professor Lu Peng e pelo doutorando Ying Zhang, ambos da Universidade do Estado da Louisiana.

A Figura 2.27 mostra as faltas de instrução médias acumuladas por mil instruções para cinco entradas em um único programa SPEC2000. Para essas entradas, a taxa de falta média para o primeiro 1,9 bilhão de instruções é muito diferente da taxa de falta média para o restante da execução.

Armadilha. *Não oferecer largura de banda de memória alta em um sistema baseado em cache.*

As caches ajudam na latência média de memória cache, mas não podem oferecer grande largura de banda de memória para uma aplicação que precisa ir até a memória principal. O arquiteto precisa projetar uma memória com grande largura de banda por trás da cache para tais aplicações. Vamos revisitar essa armadilha nos Capítulos 4 e 5.

Armadilha. *Implementar um monitor de máquina virtual em uma arquitetura de conjunto de instruções que não foi projetado para ser virtualizável.*

Nas décadas de 1970 e 1980, muitos arquitetos não tinham o cuidado de garantir que todas as instruções de leitura ou escrita de informações relacionadas com a informações de recursos de hardware fossem privilegiadas. Essa atitude *laissez-faire* causa problemas para os VMMs em todas essas arquiteturas, incluindo 80x86, que usamos aqui como exemplo.

A Figura 2.28 descreve as 18 instruções que causam problemas para a virtualização (Robin e Irvine, 2000). As duas classes gerais são instruções que

Benchmark	Faltas de L2/todas as referências à cache de dados	Faltas de L3/todas as referências à cache de dados
PERLBENCH	1%	0%
BZIP2	2%	0%
GCC	6%	1%
MCF	15%	5%
GOBMK	1%	0%
HMMER	2%	0%
SJENG	0%	0%
LIBQUANTUM	3%	0%
H264REF	1%	0%
OMNETPP	7%	3%
ASTAR	3%	1%
XALANCBMK	4%	1%
MILC	6%	1%
NAMD	0%	0%
DEALII	4%	0%
SOPLEX	9%	1%
POVRAY	0%	0%
LBM	4%	4%
SPHINX3	7%	0%

FIGURA 2.25 Taxas de falta de L2 e L3 mostradas em forma de tabela em comparação com o número de requisições de dados.

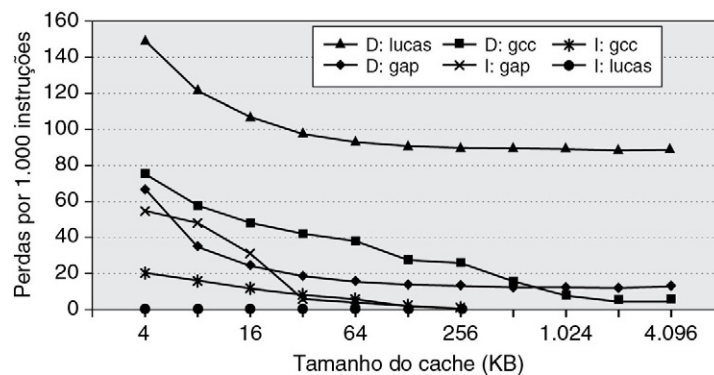



FIGURA 2.26 Faltas de instruções e dados por 1.000 instruções à medida que o tamanho da cache varia de 4 KB a 4.096 KB.

As faltas de instruções para gcc são 30.000-40.000 vezes maiores do que para o lucas e, reciprocamente, as faltas de dados para o lucas são 2-60 vezes maiores do que para o gcc. Os programas gap, gcc e lucas são do pacote de benchmark SPEC2000.

- leem registradores de controle no modo usuário, que revela que o sistema operacional está rodando em uma máquina virtual (como POPF, mencionada anteriormente); e
- verificam a proteção exigida pela arquitetura segmentada, mas presumem que o sistema operacional está rodando no nível de privilégio mais alto.

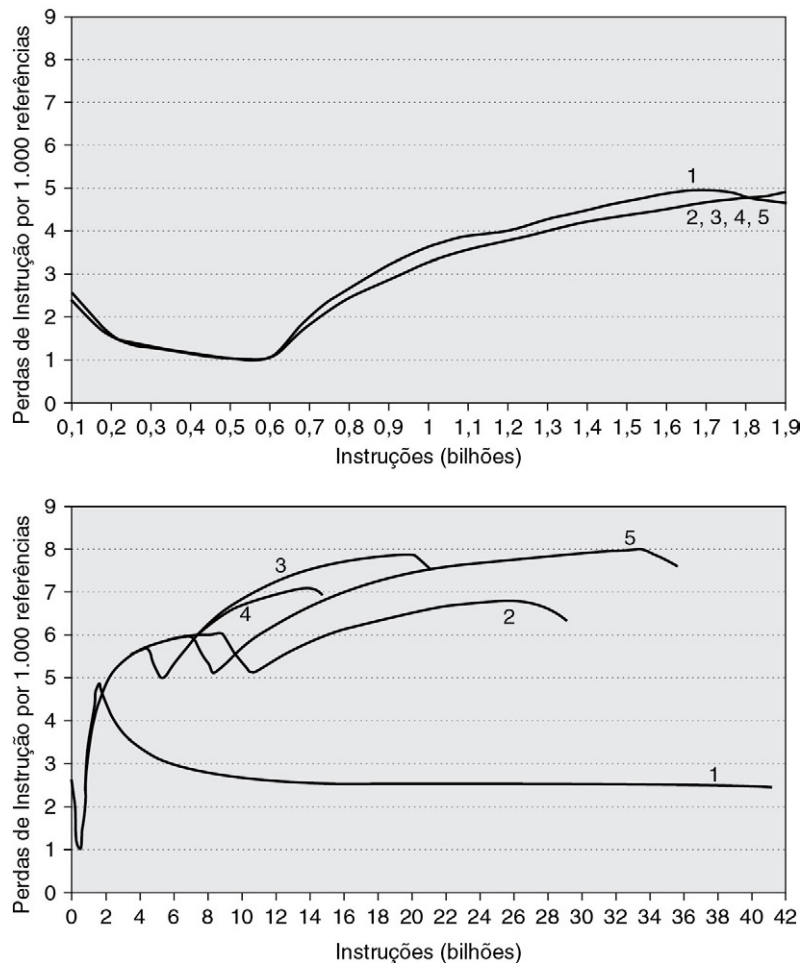



FIGURA 2.27 **Faltas de instrução por 1.000 referências para cinco entradas no benchmark perl do SPEC2000.**

Existem poucas variações nas faltas e poucas diferenças entre as cinco entradas para o primeiro 1,9 bilhão de instruções. A execução até o término mostra como as faltas variam durante a vida do programa e como elas dependem da entrada. O gráfico superior mostra as faltas médias de execução para o primeiro 1,9 bilhão de instruções, que começa em cerca de 2,5 e termina em cerca de 4,7 faltas por 1.000 referências para todas as cinco entradas. O gráfico inferior mostra as faltas médias de execução para executar até o término, que ocupa 16-41 bilhões de instruções, dependendo da entrada. Após o primeiro 1,9 bilhão de instruções, as faltas por 1.000 referências variam de 2,4-7,9, dependendo da entrada. As simulações foram para o processador Alpha usando caches L1 separadas para instruções e dados, cada qual de 64 KB em duas vias com LRU, e uma cache L2 unificada de 1 MB, mapeada diretamente.

A memória virtual também é desafiadora. Como os TLBs do 80x86 não admitem tags de ID (identificação) de processo, assim como a maioria das arquiteturas RISC, é mais dispendioso para o VMM e os SOs convidados compartilhar o TLB; cada mudança de espaço de endereço normalmente exige um esvaziamento do TLB.

A virtualização da E/S também é um desafio para o 80x86, em parte porque ele admite E/S mapeada na memória e possui instruções de E/S separadas, e em parte — o que é mais importante — porque existe um número muito grande e enorme variedade de tipos de dispositivos e drivers de dispositivo para PCs, para o VMM tratar. Os vendedores terceiros fornecem seus próprios drivers, e eles podem não virtualizar corretamente. Uma solução

Categoria de problema	Instruções 80x86 problemáticas
Acessar registradores de controle sem interceptá-los ao executar no modo usuário	Armazenar registrador da tabela de descritor global (SGDT) Armazenar registrador da tabela de descritor local (SLDT) Armazenar registrador da tabela de descritor de interrupção (SIDT) Armazenar word de status da máquina (SMSW) Push de flags (PUSHF, PUSHFD) Pop de flags (POPF, POPFD)
Ao acessar mecanismos de memória virtual no modo usuário, instruções falham nas verificações de proteção do 80x86	Carregar direitos de acesso do descritor de segmento (LAR) Carregar limite de segmento do descritor de segmento (LSL) Verificar se o descritor de segmento pode ser lido (VERR) Verificar se o descritor de segmento pode ser escrito (VERW) Pop do registrador de segmento (POP CS, POP SS, ...) Push para registrador de segmento (PUSH CS, PUSH SS, ...) Chamada distante para nível de privilégio diferente (CALL) Retorno distante para nível de privilégio diferente (RET) Salto distante para nível de privilégio diferente (JMP) Interrupção de software (INT) Armazenar registrador do seletor de segmento (STR) Mover para/de registradores de segmento (MOVE)

FIGURA 2.28 **Resumo das 18 instruções do 80x86 que causam problemas para a virtualização (Robin e Irvine, 2000).**

As cinco primeiras instruções do grupo de cima permitem que um programa no modo usuário leia um registrador de controle como os registradores da tabela de descritor, sem causar um trap. A instrução pop flags modifica um registrador de controle com informações sensíveis, mas falha silenciosamente quando está no modo usuário. A verificação de proteção da arquitetura segmentada do 80x86 é a ruína do grupo de baixo, pois cada uma dessas instruções verifica o nível de privilégio implicitamente como parte da execução da instrução quando lê um registrador de controle. A verificação pressupõe que o SO precisa estar no nível de privilégio mais alto, que não é o caso para VMs convidadas. Somente o MOVE para o registrador de segmento tenta modificar o estado de controle, e a verificação de proteção é prejudicada.

para as implementações convencionais de uma VM é carregar os drivers de dispositivo reais diretamente no VMM.

Para simplificar as implementações de VMMs no 80x86, tanto a AMD quanto a Intel propuseram extensões à arquitetura. O VT-x da Intel oferece um novo modo de execução para executar VMs, uma definição arquitetada do estado da VM, instruções para trocar VMs rapidamente e um grande conjunto de parâmetros para selecionar as circunstâncias em que um VMM precisa ser invocado. Em conjunto, o VT-x acrescenta 11 novas instruções para o 80x86. A Secure Virtual Machine (SVM) da AMD tem uma funcionalidade similar.

Depois de ativar o modo que habilita o suporte do VT-x (por meio da instrução VMXON), o VT-x oferece quatro níveis de privilégio para o SO convidado, que são inferiores em prioridade aos quatro originais. O VT-x captura todo o estado de uma máquina virtual no Virtual Machine Control State (VMCS) e depois oferece instruções indivisíveis para salvar e restaurar um VMCS. Além do estado crítico, o VMCS inclui informações de configuração para determinar quando invocar o VMM e, depois, especificamente, o que causou a invocação do VMM. Para reduzir o número de vezes que o VMM precisa ser invocado, esse modo acrescenta versões de sombra de alguns registradores sensíveis e acrescenta máscaras que verificam se os bits críticos de um registrador sensível serão alterados antes da interceptação. Para reduzir o custo da virtualização da memória virtual, a SVM da AMD acrescenta um nível de indireção adicional, chamado *tabelas de página aninhadas*. Isso torna as tabelas de página de sombra desnecessárias.

2.8 COMENTÁRIOS FINAIS: OLHANDO PARA O FUTURO

> *Ao longo dos últimos trinta anos tem havido diversas previsões do fim eminente [sic] da taxa de melhoria do desempenho dos computadores. Todas essas previsões estavam erradas, pois foram articuladas sobre suposições derrubadas por eventos subsequentes. Então, por exemplo, a falha em prever a mudança dos componentes discretos para os circuitos integrados levou a uma previsão de que a velocidade da luz limitaria a velocidade dos computadores a várias ordens de magnitude a menos do que as velocidades atuais. Provavelmente, nossa previsão sobre a barreira de memória também está errada, mas ela sugere que precisamos começar a pensar "fora da caixa".*
>
> **Wm. A. Wulf e Sally A. McKee**
>
> *Hitting the Memory Wall: Implications of the Obvious*
> **Departamento de Ciência da Computação, Universidade da Virginia (dezembro de 1994); (Esse artigo introduziu o nome *memory wall — barreira de memória*).**

A possibilidade de usar uma hierarquia de memória vem desde os primeiros dias dos computadores digitais de uso geral, no final dos anos 1940 e começo dos anos 1950. A memória virtual foi introduzida nos computadores de pesquisa, no começo dos anos 1960, e nos mainframes IBM, nos anos 1970. As caches apareceram na mesma época. Os conceitos básicos foram expandidos e melhorados ao longo do tempo para ajudar a diminuir a diferença do tempo de acesso entre a memória e os processadores, mas os conceitos básicos permanecem.

Uma tendência que poderia causar mudança significativa no projeto das hierarquias de memória é uma redução contínua de velocidade, tanto em densidade quanto em tempo de acesso nas DRAMs. Na última década, essas duas tendências foram observadas. Embora algumas melhorias na largura de banda de DRAM tenham sido alcançadas, diminuições no tempo de acesso vieram muito mais lentamente parcialmente porque, para limitar o consumo de energia, os níveis de voltagem vêm caindo. Um conceito que vem sendo explorado para aumentar a largura de banda é ter múltiplos acessos sobrepostos por banco. Isso fornece uma alternativa para o aumento do número de bancos e permite maior largura de banda. Desafios de manufatura para o projeto convencional de DRAM, que usa um capacitor em cada célula, tipicamente posicionada em uma lacuna profunda, também levaram a reduções na taxa de aumento na densidade. Já existem DRAM que não utilizam capacitores, acarretando a continuidade da melhoria da tecnologia DRAM.

Independentemente das melhorias na DRAM, a memória Flash provavelmente terá um papel maior, devido às possíveis vantagens em termos de potência e densidade. Obviamente, em PMDs, a memória Flash já substituiu os drives de disco e oferece vantagens, como "ativação instantânea", que muitos computadores desktop não fornecem. A vantagem potencial da memória Flash sobre as DRAMs — a ausência de um transistor por bit para controlar a escrita — é também seu calcanhar de Aquiles. A memória Flash deve usar ciclos de apagar-reescrever em lotes consideravelmente mais lentos. Como resultado, diversos PMDs, como o Apple iPad, usam uma memória principal SDRAM relativamente pequena combinada com Flash, que age como sistema de arquivos e como sistema de armazenamento de página para tratar a memória virtual.

Além disso, diversas abordagens totalmente novas à memória estão sendo exploradas. Elas incluem MRAMs, que usam armazenamento magnético de dados, e RAMs de mudança de fase (conhecidas como PCRAM, PCME e PRAM), que usam um vidro que pode mudar entre os estados amorfo e cristalino. Os dois tipos de memória são não voláteis e oferecem densidades potencialmente maiores do que as DRAMs. Essas ideias não são novas;

tecnologias de memória magnetorresistivas e memórias de mudança de fase estão por aí há décadas. Qualquer uma dessas tecnologias pode tornar-se uma alternativa à memória Flash atual. Substituir a DRAM é uma tarefa muito mais difícil. Embora as melhorias nas DRAMs tenham diminuído, a possibilidade de uma célula sem capacitor e outras melhorias em potencial tornam difícil apostar contra as DRAMs pelo menos durante a década seguinte.

Por alguns anos, foram feitas várias previsões sobre a chegada da barreira de memória (veja artigo citado no início desta seção), que levaria a reduções fundamentais no desempenho do processador. Entretanto, a extensão das caches para múltiplos níveis, esquemas mais sofisticados de recarregar e pré-busca, maior conhecimento dos compiladores e dos programadores sobre a importância da localidade, e o uso de paralelismo para ocultar a latência que ainda existir, vêm ajudando a manter a barreira de memória afastada. A introdução de pipelines fora de ordem com múltiplas faltas pendentes permitiu que o paralelismo em nível de instrução disponível ocultasse a latência de memória ainda existente em um sistema baseado em cache. A introdução do multithreading e de mais paralelismo no nível de thread levou isso além, fornecendo mais paralelismo e, portanto, mais oportunidades de ocultar a latência. É provável que o uso de paralelismo em nível de instrução e thread seja a principal ferramenta para combater quaisquer atrasos de memória encontrados em sistemas de cache multinível modernos.

Uma ideia que surge periodicamente é o uso de scratchpad controlado pelo programador ou outras memórias de alta velocidade, que veremos ser usadas em GPUs. Tais ideias nunca se popularizaram por várias razões: 1) elas rompem com o modelo de memória, introduzindo espaços de endereço com comportamento diferente; 2) ao contrário das otimizações de cache baseadas em compilador ou em programador (como a pré-busca), transformações de memória com scratchpads devem lidar completamente com o remapeamento a partir do espaço de endereço da memória principal para o espaço de endereço do scratchpad. Isso torna tais transformações mais difíceis e limitadas em aplicabilidade. No caso das GPUs (Cap. 4), onde memórias scratchpad locais são muito usadas, o peso de gerenciá-las atualmente recai sobre o programador.

Embora seja necessário muito cuidado quanto a prever o futuro da tecnologia da computação, a história mostrou que o uso de caches é uma ideia poderosa e altamente ampliável que provavelmente vai nos permitir continuar construindo computadores mais rápidos e garantindo que a hierarquia de memória entregue as instruções e os dados necessários para manter tais sistemas funcionando bem.

2.9 PERSPECTIVAS HISTÓRICAS E REFERÊNCIAS

Na Seção L.3 (disponível on-line), examinaremos a história das caches, memória virtual e máquinas virtuais. A IBM desempenha um papel proeminente na história dos três. As referências para leitura adicional estão incluídas nessa seção.

Estudos de caso e exercícios por Norman P. Jouppi

ESTUDOS DE CASO COM EXERCÍCIOS POR NORMAN P. JOUPPI, NAVEEN MURALIMANOHAR E SHENG LI

Estudo de caso 1: otimizando o desempenho da cache por meio de técnicas avançadas

Conceitos ilustrados por este estudo de caso

- Caches sem bloqueio
- Otimizações de compilador para as caches

- Pré-busca de software e hardware
- Cálculo de impacto do desempenho da cache sobre processadores mais complexos

A transposição de uma matriz troca suas linhas e colunas e é ilustrada a seguir:

$$\begin{bmatrix} A11 & A12 & A13 & A14 \\ A21 & A22 & A23 & A24 \\ A31 & A32 & A33 & A34 \\ A41 & A42 & A43 & A44 \end{bmatrix} \Rightarrow \begin{bmatrix} A11 & A21 & A31 & A41 \\ A12 & A22 & A32 & A42 \\ A13 & A23 & A33 & A43 \\ A14 & A24 & A34 & A44 \end{bmatrix}$$

Aqui está um loop simples em C para mostrar a transposição:

```
para (i = 0; i < 3; i++) {
 para (j = 0; j < 3; j++) {
 output[j][i] = input[i][j];
 }
}
```

Considere que as matrizes de entrada e saída sejam armazenadas na ordem principal de linha (*ordem principal de linha* significa que o índice de linha muda mais rapidamente). Suponha que você esteja executando uma transposição de precisão dupla de 256×256 em um processador com cache de dados de 16 KB totalmente associativa (de modo que não precise se preocupar com conflitos de cache) nível 1 por substituição LRU, com blocos de 64 bytes. Suponha que as faltas de cache de nível 1 ou pré-buscas exijam 16 ciclos, sempre acertando na cache de nível 2, e a cache de nível 2 possa processar uma solicitação a cada dois ciclos de processador. Suponha que cada iteração do loop interno acima exija quatro ciclos se os dados estiverem presentes na cache de nível 1. Suponha que a cache tenha uma política escrever-alocar-buscar na escrita para as faltas de escrita. Suponha, de modo não realista, que a escrita de volta dos blocos modificados de cache exija 0 ciclo.

2.1 [10/15/15/12/20] <2.2> Para a implementação simples mostrada anteriormente, essa ordem de execução seria não ideal para a matriz de entrada. Porém, a aplicação de uma otimização de troca de loops criaria uma ordem não ideal para a matriz de saída. Como a troca de loops não é suficiente para melhorar seu desempenho, ele precisa ser bloqueado.

- **a.** [10] <2.2> Que tamanho de bloco deve ser usado para preencher completamente a cache de dados com um bloco de entrada e saída?
- **b.** [15] <2.2> Como os números relativos de faltas das versões bloqueada e não bloqueada podem ser comparados se a cache de nível 1 for mapeada diretamente?
- **c.** [15] <2.2> Escreva um código para realizar transposição com um parâmetro de tamanho de bloco B que usa $B \times B$ blocos.
- **d.** [12] <2.2> Qual é a associatividade mínima requerida da cache L1 para desempenho consistente independentemente da posição dos dois arrays na memória?
- **e.** [20] <2.2> Tente as transposições bloqueada e não bloqueada de uma matriz de 256×256 em um computador. Quanto os resultados se aproximam de suas expectativas com base no que você sabe sobre o sistema de memória do computador? Explique quaisquer discrepâncias, se possível.

2.2 [10] <2.2> Suponha que você esteja reprojetando um hardware de pré-busca para o código de transposição de matriz *não bloqueado* anterior. O tipo mais simples de

hardware de pré-busca só realiza a pré-busca de blocos de cache sequenciais após uma falta. Os hardwares de pré-busca de "passos (strides) não unitários" mais complicados podem analisar um fluxo de referência de falta e detectar e pré-buscar passos não unitários. Ao contrário, a pré-busca via software pode determinar passos não unitários tão facilmente quanto determinar os passos unitários. Suponha que as pré-buscas escrevam diretamente na cache sem nenhuma "poluição" (sobrescrever dados que precisam ser usados antes que os dados sejam pré-buscados). No estado fixo do loop interno, qual é o desempenho (em ciclos por iteração) quando se usa uma unidade ideal de pré-busca de passo não unitário?

2.3 [15/20] <2.2> Com a pré-busca via software, é importante ter o cuidado de fazer com que as pré-buscas ocorram em tempo para o uso, mas também minimizar o número de pré-buscas pendentes, a fim de viver dentro das capacidades da microarquitetura e minimizar a poluição da cache. Isso é complicado pelo fato de os diferentes processadores possuírem diferentes capacidades e limitações.

a. [15] <2.2> Crie uma versão bloqueada da transposição de matriz com pré-busca via software.

b. [20] <2.2> Estime e compare o desempenho dos códigos bloqueado e não bloqueado com e sem pré-busca via software.

Estudo de caso 2: juntando tudo: sistemas de memória altamente paralelos

Conceito ilustrado por este estudo de caso

- Questões cruzadas: O projeto de hierarquias de memória

O programa apresentado na Figura 2.29 pode ser usado para avaliar o comportamento de um sistema de memória. A chave é ter temporização precisa e depois fazer com que o programa corra pela memória para invocar diferentes níveis da hierarquia. A Figura 2.29 mostra o código em C. A primeira parte é um procedimento que usa um utilitário-padrão para obter uma medida precisa do tempo de CPU do usuário; talvez esse procedimento tenha de mudar para funcionar em alguns sistemas. A segunda parte é um loop aninhado para ler e escrever na memória em diferentes passos e tamanhos de cache. Para obter tempos de cache precisos, esse código é repetido muitas vezes. A terceira parte temporiza somente o overhead do loop aninhado, de modo que possa ser subtraído dos tempos medidos em geral para ver quanto tempo os acessos tiveram. Os resultados são enviados para o formato de arquivo .csv, para facilitar a importação em planilhas. Você pode ter de mudar CACHE_MAX, dependendo da pergunta a que estiver respondendo e do tamanho da memória no sistema que estiver medindo. A execução do programa no modo monousuário ou pelo menos sem outras aplicações ativas dará resultados mais coerentes. O código mostrado na Figura 2.29 foi derivado de um programa escrito por Andrea Dusseau, da U.C. Berkeley, baseado em uma descrição detalhada encontrada em Saavedra-Barrera (1992). Ele foi modificado para resolver uma série de problemas com máquinas mais modernas e para executar sob o Microsoft Visual C++. Ele pode ser baixado em <www.hpl.hp.com/research/cacti/aca_ch2_cs2.c>.

O programa mostrado anteriormente considera que os endereços de memória rastreiam os endereços físicos, o que é verdadeiro em algumas máquinas que usam caches endereçadas virtualmente, como o Alpha 21264. Em geral, os endereços virtuais costumam acompanhar os endereços físicos logo depois da reinicialização, de modo que você pode ter de reinicializar a máquina a fim de conseguir linhas suaves nos seus resultados. Para fazer os exercícios, considere que os tamanhos de todos os componentes da hierarquia de memória sejam potências de 2. Considere ainda que o tamanho da página é muito maior do que o tamanho de um bloco em uma cache de segundo nível (se houver uma) e que o tamanho

```
#include "stdafx.h"
#include <stdio.h>
#include <time.h>
#define ARRAY_MIN (1024) /* 1/4 smallest cache */
#define ARRAY_MAX (4096*4096) /* 1/4 largest cache */

int x[ARRAY_MAX]; /* array going to stride through */
double get_seconds() { /* routine to read time in seconds */
   __time64_t ltime;
   _time64( &ltime );
   return (double) ltime;
}
int label(int i) {/* generate text labels */
   if (i<1e3) printf("%1dB,",i);
   else if (i<1e6) printf("%1dK,",i/1024);
   else if (i<1e9) printf("%1dM,",i/1048576);
   else printf("%1dG,",i/1073741824);
   return 0;
}
int _tmain(int argc, _TCHAR* argv[]) {
int register nextstep, i, index, stride;
int csize;
double steps, tsteps;
double loadtime, lastsec, sec0, sec1, sec; /* timing variables */

/* Initialize output */
printf(" ,");
for (stride=1; stride <= ARRAY_MAX/2; stride=stride*2)
label(stride*sizeof(int));
printf("\n");

/* Main loop for each configuration */
for (csize=ARRAY_MIN; csize <= ARRAY_MAX; csize=csize*2) {
    label(csize*sizeof(int)); /* print cache size this loop */
    for (stride=1; stride <= csize/2; stride=stride*2) {

       /* Lay out path of memory references in array */
       for (index=0; index < csize; index=index+stride)
          x[index] = index + stride; /* pointer to next */
       x[index-stride] = 0; /* loop back to beginning */

       /* Wait for timer to roll over */
       lastsec = get_seconds();
       sec0 = get_seconds(); while (sec0 == lastsec);

       /* Walk through path in array for twenty seconds */
       /* This gives 5% accuracy with second resolution */
       steps = 0.0; /* number of steps taken */
       nextstep = 0; /* start at beginning of path */
       sec0 = get_seconds(); /* start timer */
          { /* repeat until collect 20 seconds */
             (i=stride;i!=0;i=i-1) { /* keep samples same */
                nextstep = 0;
                do nextstep = x[nextstep]; /* dependency */
                while (nextstep != 0);
          }
          steps = steps + 1.0; /* count loop iterations */
          sec1 = get_seconds(); /* end timer */
       } while ((sec1 - sec0) < 20.0); /* collect 20 seconds */
       sec = sec1 - sec0;

       /* Repeat empty loop to loop subtract overhead */
       tsteps = 0.0; /* used to match no. while iterations */
       sec0 = get_seconds(); /* start timer */
           { /* repeat until same no. iterations as above */
               (i=stride;i!=0;i=i-1) { /* keep samples same */
                  index = 0;
                  do index = index + stride;
                  while (index < csize);
           }
           tsteps = tsteps + 1.0;
           sec1 = get_seconds(); /* - overhead */
       } while (tsteps<steps); /* until = no. iterations */
       sec = sec - (sec1 - sec0);
       loadtime = (sec*1e9)/(steps*csize);
       /* write out results in .csv format for Excel */
       printf("%4.1f,", (loadtime<0.1) ? 0.1 : loadtime);
     }; /* end of inner for loop */
     printf("\n");
   }; /* end of outer for loop */
   return 0;
}
```

FIGURA 2.29 Programa em C para avaliar os sistemas de memória.

de um bloco de cache de segundo nível é maior ou igual ao tamanho de um bloco em uma cache de primeiro nível. Um exemplo da saída do programa é desenhado na Figura 2.30, com a chave listando o tamanho do array que é exercitado.

2.4 [12/12/12/10/12] <2.6> Usando os resultados do programa de exemplo na Figura 2.30:

a. [12] <2.6> Quais são o tamanho geral e o tamanho de bloco da cache de segundo nível?

b. [12] <2.6> Qual é a penalidade de falta da cache de segundo nível?

c. [12] <2.6> Qual é a associatividade da cache de segundo nível?

d. [10] <2.6> Qual é o tamanho da memória principal?

e. [12] <2.6> Qual será o tempo de paginação se o tamanho da página for de 4 KB?

2.5 [12/15/15/20] <2.6> Se necessário, modifique o código na Figura 2.29 para medir as seguintes características do sistema. Desenhe os resultados experimentais com o tempo decorrido no eixo *y* e o stride da memória no eixo *x*. Use escalas logarítmicas para os dois eixos e desenhe uma linha para cada tamanho de cache.

a. [12] <2.6> Qual é o tamanho de página do sistema?

b. [15] <2.6> Quantas entradas existem no *translation lookaside buffer* (TLB)?

c. [15] <2.6> Qual é a penalidade de falta para o TLB?

d. [20] <2.6> Qual é a associatividade do TLB?

2.6 [20/20] <2.6> Em sistemas de memória de multiprocessadores, níveis inferiores da hierarquia de memória podem não ser capazes de ser saturados por um único processador, mas devem ser capazes de ser saturados por múltiplos processadores trabalhando juntos. Modifique o código na Figura 2.29 e execute múltiplas cópias ao mesmo tempo. Você pode determinar:

a. [20] <2.6> Quantos processadores reais estão no seu sistema de computador e quantos processadores de sistema são só contextos mutithread adicionais?

b. [20] <2.6> Quantos controladores de memória seu sistema tem?

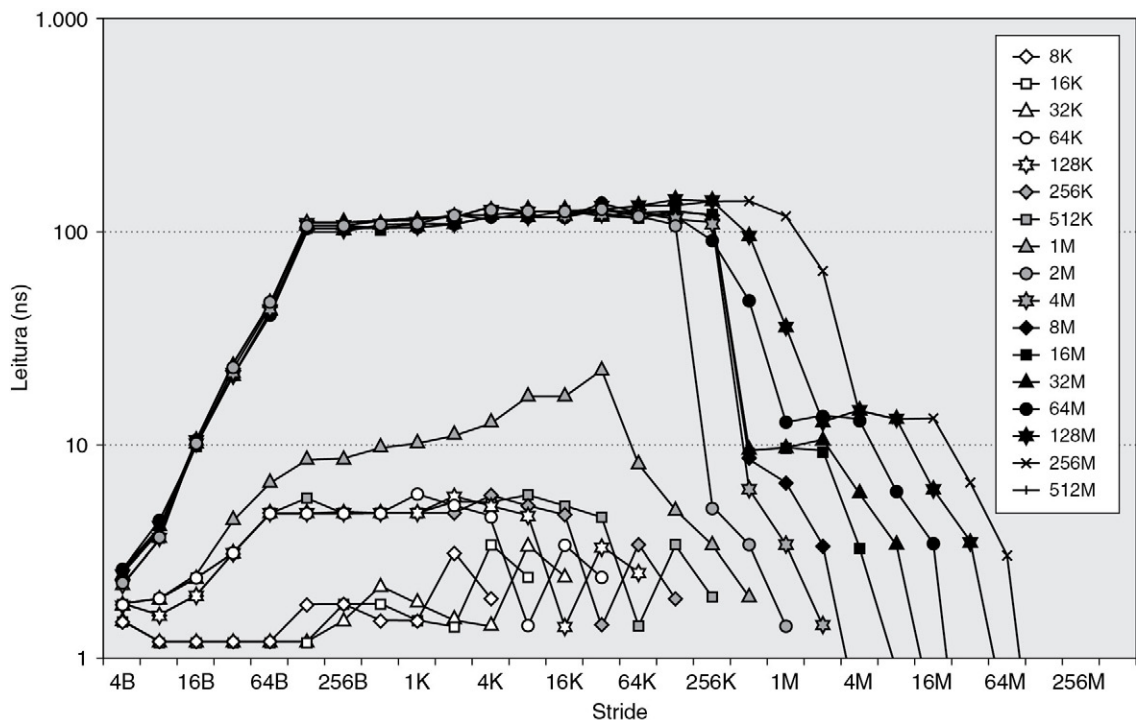



FIGURA 2.30 Resultados de exemplo do programa da Figura 2.29.

2.7 [20] <2.6> Você pode pensar em um modo de testar algumas das características de uma cache de instrução usando um programa? *Dica:* O compilador pode gerar grande número de instruções não óbvias de um trecho de código. Tente usar instruções aritméticas simples de comprimento conhecido da sua arquitetura de conjunto de instruções (ISA).

Exercícios

2.8 [12/12/15] <2.2> As perguntas a seguir investigam o impacto de caches pequenas e simples usando CACTI e presumindo uma tecnologia de 65 nm (0,065 μm). (O CACTI está disponível on-line em <http://quid.hpl.hp.com:9081/cacti/>).

- **a.** [12] <2.2> Compare os tempos de acesso das caches de 64 KB com blocos de 64 bytes em um único banco. Quais são os tempos de acesso relativos de caches associativas por conjunto de duas e quatro vias em comparação com uma organização mapeada de modo diferente?
- **b.** [12] <2.2 Compare os tempos de acesso de caches associativas por conjunto de quatro vias com blocos de 64 bytes e um único banco. Quais são os tempos relativos de acesso de caches de 32 KB e 64 KB em comparação com uma cache de 16 KB?
- **c.** [15] <2.2> Para uma cache de 64 KB, encontre a associatividade de cache entre 1 e 8 com o menor tempo médio de acesso à memória, dado que as faltas por instrução para certa carga de trabalho é de 0,00664 para mapeamento direto, 0,00366 para associativas por conjunto de duas vias, 0,000987 para associativas por conjunto de quatro vias e 0,000266 para cache de associativas por conjunto de oito vias. Em geral existem 0,3 referência de dados por instrução. Suponha que as faltas de cache levem 10 ns em todos os modelos. Para calcular o tempo de acerto em ciclos, suponha a saída de tempo de ciclo usando CACTI, que corresponde à frequência máxima em que uma cache pode operar sem "bolhas" no pipeline.

2.9 [12/15/15/10] <2.2> Você está investigando os possíveis benefícios de uma cache L1 com previsão de via. Considere que a cache de dados L1 com 64 KB, associativas por conjunto com duas vias e único banco, seja atualmente o limitador do tempo de ciclo. Como organização de cache alternativa, você está considerando uma cache com previsão de via modelado como uma cache mapeada diretamente de 64 KB, com 80% de exatidão na previsão. A menos que indicado de outra forma, considere um acesso à via mal previsto que chegue à cache utilizando mais um ciclo. Considere as taxas de falta e as penalidades de falta da Questão 2.8, item (c).

- **a.** [12] <2.2> Qual é o tempo médio de acesso à memória da cache atual (em ciclos) contra a cache com previsão de via?
- **b.** [15] <2.2> Se todos os outros componentes pudessem operar com o tempo de ciclo de clock com previsão de via mais rápido (incluindo a memória principal), qual seria o impacto sobre o desempenho de usar a cache com previsão de via?
- **c.** [15] <2.2> As caches com previsão de via normalmente só têm sido usadas para caches de instrução que alimentam uma fila de instrução ou buffer. Imagine que você deseje experimentar a previsão de via em uma cache de dados. Suponha que você tenha 80% de exatidão na previsão e que as operações subsequentes (p. ex., acesso da cache de dados de outras instruções, dependentes das operações) sejam emitidas pressupondo uma previsão de via correta. Assim, um erro de previsão de via necessita de um esvaziamento de pipe e interceptação da repetição, o que exige 15 ciclos. A mudança no tempo

médio de acesso à memória por instrução de carregamento com previsão de via na cache de dados é positiva ou negativa? Quanto?

d. [10] <2.2> Como alternativa à previsão de via, muitas caches associativas grandes L2 serializam o acesso a tags e dados, de modo que somente o array do conjunto de dados exigido precisa ser ativado. Isso economiza energia, mas aumenta o tempo de acesso. Use a interface Web detalhada do CACTI para uma cache associativa por conjunto de quatro vias, com 1 MB e processo de 0,065 μm com blocos de 64 bytes, 144 bits lidos, um banco, somente uma porta de leitura/escrita e tags de 30 bits. Quais são a razão de energias de leitura dinâmica total por acesso e a razão dos tempos de acesso para serializar o acesso a tags e dados em comparação com o acesso paralelo?

2.10 [10/12] <2.2> Você recebeu a tarefa de investigar o desempenho relativo de uma cache de dados nível 1 em banco contra outro em pipeline para um novo microprocessador. Considere uma cache associativa por conjunto de duas vias e 64 KB, com blocos de 64 bytes. A cache em pipeline consistiria em dois estágios de pipe, semelhante à cache de dados do Alpha 21264. Uma implementação em banco consistiria em dois bancos associativos por conjunto de duas vias e 32 KB. Use o CACTI e considere uma tecnologia de 90 nm (0,09 μm) na resposta às perguntas a seguir.

a. [10] <2.2> Qual é o tempo de ciclo da cache em comparação com o seu tempo de acesso? Quantos estágios de pipe a cache ocupará (até duas casas decimais)?

b. [12] <2.2> Compare a energia de leitura dinâmica total e de área por acesso do projeto com pipeline com o projeto com bancos. Diga qual ocupa menos área e qual requer mais potência, e explique o porquê.

2.11 [12/15] <2.2> Considere o uso de palavra crítica primeiro e o reinício antecipado em faltas de cache L2. Suponha uma cache L2 de 1 MB com blocos de 64 bytes e uma via de recarregar com 16 bytes de largura. Suponha que o L2 possa ser escrito com 16 bytes a cada quatro ciclos de processador, o tempo para receber o primeiro bloco de 16 bytes do controlador de memória é de 120 ciclos, cada bloco adicional de 16 bytes da memória principal requer 16 ciclos, e os dados podem ser enviados diretamente para a porta de leitura da cache L2. Ignore quaisquer ciclos para transferir a requisição de falta para a cache L2 e os dados requisitados para a cache L1.

a. [12] <2.2> Quantos ciclos levaria para atender uma falta de cache L2 com e sem palavra crítica primeiro e reinício antecipado?

b. [15] <2.2> Você acha que a palavra crítica primeiro e o reinício antecipado seriam mais importantes para caches L1 e L2? Que fatores contribuiriam para sua importância relativa?

2.12 [12/12] <2.2> Você está projetando um buffer de escrita entre uma cache write-through L1 e uma cache write-back L2. O barramento de dados de escrita da cache de L2 tem 16 bytes de largura e pode realizar escrita em um endereço de cache independente a cada quatro ciclos do processador.

a. [12] <2.2> Quantos bytes de largura cada entrada do buffer de escrita deverá ter?

b. [15] <2.2> Que ganho de velocidade poderia ser esperado no estado constante usando um write buffer merge em vez de um buffer sem mesclagem quando a memória estiver sendo zerada pela execução de armazenamentos de 64 bits, se todas as outras instruções puderem ser emitidas em paralelo com os armazenamentos e os blocos estiverem presentes na cache L2?

c. [15] <2.2> Qual seria o efeito das possíveis faltas em L1 no número de entradas necessárias de buffer de escrita para sistemas com caches com blocos e sem blocos?

2.13 [10/10/10] <2.3> Considere um sistema de desktop com um processador conectado a uma DRAM de 2 GB com *código de correção de erro* (ECC). Suponha que exista somente um canal de memória com largura de 72 bits para 64 bits para dados e 8 bits para ECC.

a. [10] <2.3> Quantos chips DRAM estão na DIMM, se forem usados chips de DRAM 1 GB, e quantas E/S de dados cada DRAM deve ter se somente uma DRAM se conecta a cada pino de dados da DIMM?

b. [10] <2.3> Que duração de burst é necessária para suportar blocos de cache L2 de 32 KB?

c. [10] <2.3> Calcule o pico de largura de banda para as DIMMs DDR2-667 e DDR2-533 para leituras de uma página ativa excluindo o overhead do ECC.

2.14 [10/10] <2.3> Um exemplo de diagrama de temporização SDRAM DDR2 aparece na Figura 2.31. t_{RCD} é o tempo exigido para ativar uma linha em um banco, enquanto a latência CAS (CL) é o número de ciclos exigidos para ler uma coluna em uma linha. Considere que a RAM esteja em um DIMM DDR2 com ECC tendo 72 linhas de dados. Considere também extensões de burst de 8 que leem 8 bits por linha de dados, ou um total de 64 bytes do DIMM. Considere t_{RCD} = CAS (ou CL)* frequência_clock e frequência_clock = transferências_por_segundo/2. A latência no chip em uma falta de cache através dos níveis 1 e 2 e de volta, sem incluir o acesso à DRAM, é de 20 ns.

a. [10] <2.3> Quanto tempo é necessário da apresentação do comando de ativação até que o último bit de dados solicitado das transições de DRAM de válido para inválido para a DIMM DDR2-667 de 1 GB CL-5? Suponha que, para cada requisição, fazemos a pré-busca automaticamente de outra linha de cache adjacente na mesma.

b. [10] <2.3> Qual é a latência relativa quando usamos a DIMM DDR2-667 de uma leitura requerendo um banco ativo em vez de um para uma página já aberta, incluindo o tempo necessário para processar a falta dentro do processador?

2.15 [15] <2.3> Considere que um DIM DDR2-667 de 2 GB com CL = 5 esteja disponível por US$ 130 e uma DIMM DDR2-533 de 2 GB com CL = 4 esteja disponível por US$ 100. Considere o desempenho do sistema usando as DIMMs DDR2-667 e DDR2-533 em uma carga de trabalho com 3,33 faltas em L2 por 1 K instruções, e suponha que 80% de todas as leituras de DRAM exijam uma ativação. Qual é o custo-desempenho de todo o sistema quando usamos as diferentes DIMMs, presumindo que somente uma falta em L2 seja pendente em dado momento e um núcleo em ordem com uma CPI de 1,5 não inclua tempo de acesso à memória para falta de cache?

2.16 [12] <2.3> Você está provisionando um servidor com CMP de oito núcleos de 3 GHz, que pode executar uma carga de trabalho com uma CPI geral de 2,0 (supondo que os recarregamentos de falta de cache L2 não sejam atrasados). O tamanho de linha da cache L2 é de 32 bytes. Supondo que o sistema use DIMMs

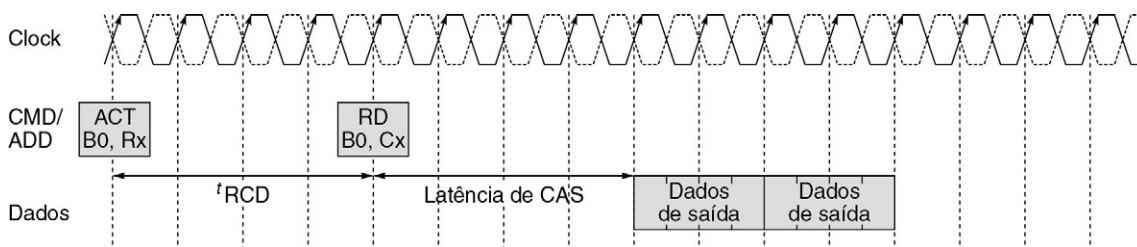



FIGURA 2.31 Diagrama de temporização da SDRAM DDR2.

DDR2-667, quantos canais independentes de memória devem ser provisionados para que o sistema não seja limitado pela largura de banda da memória se a largura de banda necessária for algumas vezes o dobro da média? As cargas de trabalho incorrem, em média, em 6,67 faltas de L2 por 1 K instruções.

2.17 [12/12] <2.3> Grande quantidade (mais de um terço) de potência de DRAM pode ser devida à ativação da página (<http://download.micron.com/pdf/technotes/ddr2/TN4704.pdf> e <www.micron.com/systemcalc>). Suponha que você esteja montando um sistema com 2 GB de memória usando DRAMs DDR2 de 2 GB x8 com oito bancos ou DRAMs de 1 GB × 8 com oito bancos, as duas com a mesma classe de velocidade. Ambas utilizam tamanho de página de 1 KB, e o tamanho da linha de cache do último nível é de 64 bytes. Suponha que as DRAMs que não estão ativas estejam em stand-by pré-carregado e dissipem uma potência insignificante. Suponha que o tempo para a transição de stand-by para ativo não seja significativo.

a. [12] <2.3> Qual tipo de DRAM você acha que resultaria em menor potência? Explique o porquê.

b. [12] <2.3> Como uma DIMM de 2GB composta de DRAMs DDR2 de 1 GB x8 se compara em termos de potência com uma DIMM com capacidade similar composta DRAM DDR2 de 1 GB x4?

2.18 [20/15/12] <2.3> Para acessar dados de uma DRAM típica, primeiro temos de ativar a linha apropriada. Suponha que isso traga uma página inteira com tamanho de 8 KB para o buffer de linha. Então, nós selecionamos determinada coluna do buffer de linha. Se acessos subsequentes à DRAM forem feitos à mesma página, poderemos pular o passo da ativação. Caso contrário, precisaremos fechar a página atual e pré-carregar as linhas de bit para a próxima ativação. Outra política popular de DRAM é fechar proativamente uma página e pré-carregar linhas de bits assim que um acesso for encerrado. Suponha que todas as leituras ou escritas para a DRAM sejam de 64 bytes e a latência do barramento DDR (dados de saída na Figura 2.30) para enviar 512 bits seja Tddr.

a. [20] <2.3> Considerando a DDR2-667, se ela levar cinco ciclos para pré-carregar, cinco ciclos para se ativar e quatro ciclos para ler uma coluna, para que valor da taxa de acerto do buffer de linha (*r*) você vai escolher uma política no lugar da outra para obter o melhor tempo de acesso? Suponha que cada acesso à DRAM seja separado por tempo suficiente para terminar um novo acesso aleatório.

b. [15] <2.3> Se 10% dos acessos totais à DRAM acontecessem back to back ou continuamente, sem intervalo de tempo, como sua decisão mudaria?

c. [12] <2.3> Calcule a diferença na energia média da DRAM por acesso entre as duas políticas usando a taxa de acerto de buffer de linhas calculada anteriormente. Suponha que o pré-carregamento requeira 2 nJ e a ativação requeira 4 nJ, e que 100 pJ/bit sejam necessários para ler ou escrever a partir do buffer de linha.

2.19 [15] <2.3> Sempre que um computador está inativo, podemos colocá-lo em stand-by (onde a DRAM ainda está ativa) ou deixá-lo hibernar. Suponha que, para a hibernação, tenhamos que copiar somente o conteúdo da DRAM para um meio não volátil, como uma memória Flash. Se ler ou escrever em uma linha de cache de tamanho 64 bytes para Flash requerer 2,56 μJ e a DRAM requerer 0,5 nJ, e se a potência em estado inativo para a DRAM for de 1,6 W (para 8 GB), quanto tempo um sistema deverá permanecer inativo para se beneficiar da hibernação? Suponha uma memória principal com 8 GB de tamanho.

2.20 [10/10/10/10/10] <2.4> As máquinas virtuais (VMs) possuem o potencial de incluir muitas capacidades benéficas aos sistemas de computador, resultando, por

exemplo, em custo total da posse (Total Cost of Ownership — TCO) melhorado ou disponibilidade melhorada. As VMs poderiam ser usadas para fornecer as capacidades a seguir? Caso afirmativo, como elas poderiam facilitar isso?

a. [10] <2.4> Testar aplicações em ambientes de produção usando máquinas de desenvolvimento?

b. [10] <2.4> Reimplementação rápida de aplicações em caso de desastre ou falha?

c. [10] <2.4> Desempenho mais alto nas aplicações com uso intensivo das E/S?

d. [10] <2.4> Isolamento de falha entre aplicações diferentes, resultando em maior disponibilidade dos serviços?

e. [10] <2.4> Realizar manutenção de software nos sistemas enquanto as aplicações estão sendo executadas sem interrupção significativa?

2.21 [10/10/12/12] <2.4> As máquinas virtuais podem perder desempenho devido a uma série de eventos, como a execução de instruções privilegiadas, faltas de TLB, traps e E/S. Esses eventos normalmente são tratados no código do sistema. Assim, um modo de estimar a lentidão na execução sob uma VM é a porcentagem de tempo de execução da aplicação no sistema contra o modo usuário. Por exemplo, uma aplicação gastando 10% de sua execução no modo do sistema poderia retardar em 60% quando fosse executada em uma VM. A Figura 2.32 lista o desempenho inicial de diversas chamadas de sistema sob execução nativa, virtualização pura e paravirtualização para LMbench usando Xen em um sistema Itanium com tempos medidos em microssegundos (cortesia de Matthew Chapman da Universidade de New South Wales).

a. [10] <2.4> Que tipos de programa poderiam ter maior lentidão quando executados sob VMs?

b. [10] <2.4> Se a lentidão fosse linear, como função do tempo do sistema, dada a lentidão anterior, quão mais lentamente um programa será executado se estiver gastando 20% de sua execução no tempo do sistema?

c. [12] <2.4> Qual é a lentidão média das funções na tabela sob a virtualização pura e paravirtualização?

d. [12] <2.4> Quais funções da tabela possuem os menores atrasos? Qual você acha que poderia ser a causa disso?

2.22 [12] <2.4> A definição de uma máquina virtual de Popek e Goldberg estabelecia que ela seria indistinguível de uma máquina real, exceto por seu desempenho. Neste exercício, usaremos essa definição para descobrir se temos acesso à execução nativa

Benchmark	Nativa	Pura	Para
Null call	0,04	0,96	0,5
Null I/O	0,27	6,32	2,91
Stat	1,1	10,69	4,14
Open/close	1,99	20,43	7,71
Install sighandler	0,33	7,34	2,89
Handle signal	1,69	19,26	2,36
Fork	56,00	513,00	164,00
Exec	316,00	2.084,00	578,00
Fork + exec sh	1.451,00	7.790,00	2.360,00

FIGURA 2.32 Desempenho inicial de diversas chamadas do sistema sob execução nativa, virtualização pura e paravirtualização.

em um processador ou se estamos executando em uma máquina virtual. A tecnologia VT-x da Intel, efetivamente, oferece um segundo conjunto de níveis de privilégio para o uso da máquina virtual. O que uma máquina virtual sendo executada sobre outra máquina virtual precisaria fazer, considerando a tecnologia VT-x?

2.23 [20/25] <2.4> Com a adoção do suporte à virtualização na arquitetura x86, as máquinas virtuais estão ativamente evoluindo e se popularizando. Compare e contraste a virtualização das tecnologias Intel VT-x e da AMD AMD-V. (Informações sobre a AMD-V podem ser encontradas em <http://sites.amd.com/us/business/it-solutions/virtualization/Pages/resources.aspx>.)

a. [20] <2.4> Qual delas proporcionaria maior desempenho para aplicações intensas no uso da memória com grande necessidade de memória?

b. [25] <2.4> Informações sobre o suporte IOMMU para E/S virtualizadas da AMD podem ser encontradas em <http://developer.amd.com/documentation/articles/pages/892006101.aspx>. O que a tecnologia de virtualização e uma unidade de gerenciamento de entrada/saída (IOMMU) podem fazer para melhorar o desempenho das E/S virtualizadas?

2.24 [30] <2.2, 2.3> Como o paralelismo em nível de instrução também pode ser explorado efetivamente nos processadores superescalares em ordem e VLIWs com especulação, um motivo importante para montar um processador superescalar fora de ordem (out-of-order-OOO) é a capacidade de tolerar a latência de memória imprevisível causada por faltas de cache. Logo, você pode pensar no hardware que suporta a emissão OOO como fazendo parte do sistema de memória! Veja a planta baixa do Alpha 21264 na Figura 2.33 para descobrir a área relativa das filas de emissão e mapeadores de inteiros e ponto flutuante contra as caches. As filas programam as instruções por emissão, e

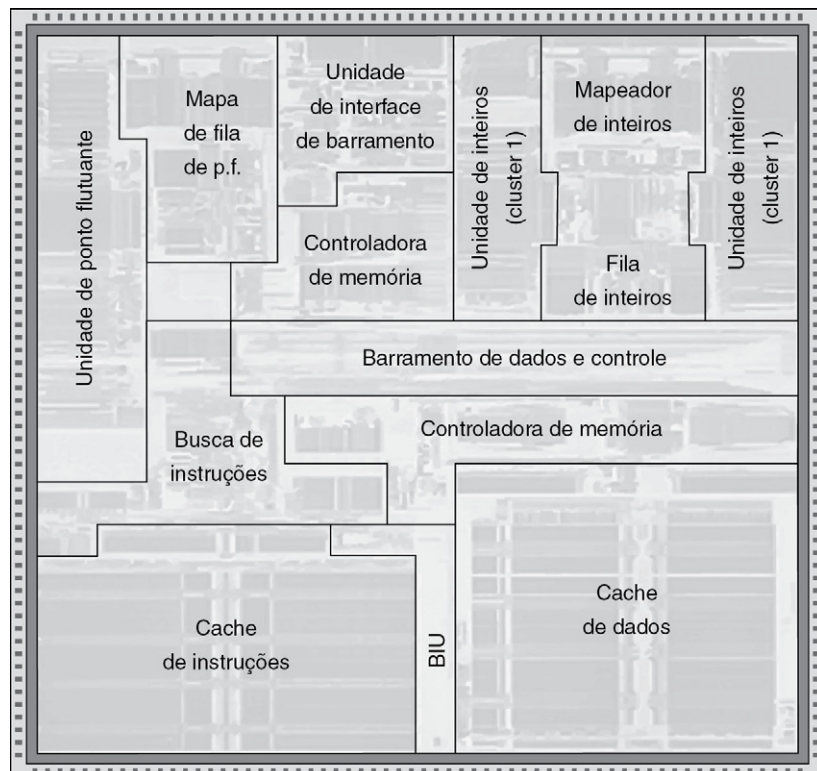



FIGURA 2.33 Planta baixa do Alpha 21264 (Kessler, 1999).

os mapeadores renomeiam os especificadores de registrador. Logo, estes são acréscimos necessários para dar suporte à emissão OOO. O 21264 só possui caches de dados e instruções de nível 1 no chip, e ambos são associativas por conjunto com duas vias e 64 KB. Use um simulador superescalar OOO, como o Simplescalar (www.cs.wisc.edu/~mscalar/simplescalar.html) nos benchmarks com uso intensivo de memória para descobrir quanto desempenho é perdido se a área das filas de emissão e mapeadores for usada para a área da cache de dados de nível 1 adicional em um processador superescalar em ordem, em vez da emissão OOO em um modelo do 21264. Certifique-se de que os outros aspectos da máquina sejam os mais semelhantes possíveis para tornar a comparação justa. Ignore qualquer aumento no tempo de acesso ou ciclo a partir de caches maiores e os efeitos da cache de dados maior na planta baixa do chip. (Observe que essa comparação não será totalmente justa, pois o código não terá sido programado para o processador fora de ordem pelo compilador.)

2.25 [20/20/20] <2.6> O analisador de desempenho VTune da Intel pode ser usado para realizar muitas medições do comportamento da cache. Uma versão gratuita para avaliação do VTune para Windows e Linux pode ser encontrada em <http://software.intel.com/enus/articles/intel-vtune-amplifier-xe/>. O programa (aça.ch2.cs2.c) usado no Estudo de Caso 2 foi modificado para funcionar prontamente com o VTune em Microsoft Visual C++. O programa pode ser obtido em <www.hpl.hp.com/research/cacti/aca_ch2_cs2_vtune.c>. Foram adicionadas funções especiais do VTunes para excluir a inicialização e o overhead de loop durante o processo de análise de desempenho. Instruções detalhadas da configuração do VTunes são dadas na seção README do programa. O programa permanece em loop por 20 segundos para cada configuração. No experimento a seguir você poderá descobrir os efeitos do tamanho dos dados sobre a cache e sobre o desempenho geral do processador. Execute o programa no VTube em um processador Intel com os tamanhos de conjunto de dados de 8 KB, 128 KB, 4 MB e 32 MB, e mantenha um passo de 64 bytes (um passo de uma linha de cache nos processadores Intel i7). Colete estatísticas sobre o desempenho geral e das caches de dados L1, L2 e L3.

a. [20] <2.6> Liste o número de faltas por 1 K instruções da cache de dados L1, L2 e L3 para cada tamanho de conjunto de dados e seu modelo e velocidade de processador. Com base nos resultados, o que você pode dizer sobre os tamanhos de cache de dados L1, caches L2 e L3 do seu processador? Explique suas observações.

b. [20] <2.6> Liste as *instruções por clock* (IPC) para cada tamanho de conjunto de dados e seu modelo e velocidade de processador. Com base nos resultados, o que você pode dizer sobre as penalidades de falta de L1, L2 e L3 do seu processador? Explique suas observações.

c. [20] <2.6> Execute o programa no VTune com tamanho de conjunto de dados de 8 KB e 125 KB em um processador OOO Intel. Liste o número de faltas da cache de dados L1 e cache L2 por 1 K instruções e a CPI para as duas configurações. O que você pode dizer sobre a eficácia das técnicas de ocultamento de latência da memória em processadores OOO de alto desempenho? *Dica:* Você precisa encontrar a latência de falta da cache de dados L1 do seu processador. Para processadores Intel i7 recentes, ela é de aproximadamente 11 ciclos.

Paralelismo em nível de instrução e sua exploração

"*Quem é o primeiro?*"
"*América.*"
"*Quem é o segundo?*"
"*Senhor, não existe segundo.*"

Diálogo entre dois observadores da corrida de veleiro chamada "Copa da América", realizada de alguns em alguns anos — a inspiração para John Cocke nomear o processador em pesquisa da IBM como "América". Esse processador foi o precursor da série RS/6000 e o primeiro microprocessador superescalar.

3.1 PARALELISMO EM NÍVEL DE INSTRUÇÃO: CONCEITOS E DESAFIOS

Desde cerca de 1985, todos os processadores utilizam pipelining para sobrepor a execução de instruções e melhorar o desempenho. Essa potencial sobreposição das instruções é chamada *paralelismo em nível de instrução* (Instruction-Level Parallelism — ILP), pois as

instruções podem ser avaliadas em paralelo. Neste capítulo e no Apêndice H, veremos grande variedade de técnicas para ampliar os conceitos básicos de pipelining, aumentando a quantidade de paralelismo explorada entre as instruções.

Este capítulo está em um nível consideravelmente mais avançado do que o material básico sobre pipelining, no Apêndice C. Se você não estiver acostumado com as ideias desse apêndice, deverá revê-lo antes de se aventurar por este capítulo.

Começaremos o capítulo examinando a limitação imposta pelos hazards de dados e hazards de controle, e depois passaremos para o tópico relacionado com o aumento da capacidade do compilador e do processador de explorar o paralelismo. Essas seções introduzirão grande quantidade de conceitos, que acumulamos no decorrer deste capítulo e do Capítulo 4. Embora parte do material mais básico deste capítulo pudesse ser entendida sem todas as ideias das duas primeiras seções, esse material básico é importante para outras seções deste capítulo.

Existem duas abordagens altamente separáveis para explorar o ILP: 1) uma que conta com o hardware para ajudar a descobrir e explorar o paralelismo dinamicamente e 2) uma que conta com a tecnologia de software para encontrar o paralelismo, estaticamente, no momento da compilação. Os processadores usando a abordagem dinâmica, baseada no hardware, incluindo a série Core da Intel, dominam os mercados de desktop e servidor. No mercado de dispositivos pessoais móveis, no qual muitas vezes a eficiência energética é o objetivo principal, os projetistas exploram níveis inferiores de paralelismo em nível de instrução. Assim, em 2011, a maior parte dos processadores para o mercado de PMDs usa abordagens estáticas, como veremos no ARM Cortex-A8. Entretanto, processadores futuros (p. ex., o novo ARM Cortex-A9) estão usando abordagens dinâmicas. Abordagens agressivas baseadas em compilador foram tentadas diversas vezes desde os anos 1980 e mais recentemente na série Intel Itanium. Apesar dos enormes esforços, tais abordagens não obtiveram sucesso fora da estreita gama de aplicações científicas.

Nos últimos anos, muitas das técnicas desenvolvidas para uma abordagem têm sido exploradas dentro de um projeto que conta basicamente com a outra. Este capítulo introduz os conceitos básicos e as duas abordagens. Uma discussão sobre as limitações das abordagens ILP é incluída neste capítulo, e foram tais limitações que levaram diretamente ao movimento para o multicore. Entender as limitações ainda é importante para equilibrar o uso de ILP e paralelismo em nível de thread.

Nesta seção, discutiremos recursos de programas e processadores que limitam a quantidade de paralelismo que pode ser explorada entre as instruções, além do mapeamento crítico entre a estrutura do programa e a estrutura do hardware, que é a chave para entender se uma propriedade do programa realmente limitará o desempenho e em quais circunstâncias.

O valor do CPI (ciclos por instruções) para um processador em pipeline é a soma do CPI base e todas as contribuições de stalls:

$$\begin{aligned}\text{CPI de pipeline } &= \text{CPI de pipeline ideal} + \text{Stalls estruturais} \\ &\quad + \text{Stalls de hazard de dados} + \text{Stalls de controle}\end{aligned}$$

O *CPI de pipeline ideal* é uma medida do desempenho máximo que pode ser obtida pela implementação. Reduzindo cada um dos termos do lado direito, minimizamos o CPI de pipeline geral ou, como alternativa, aumentamos o valor do IPC (instruções por clock). A equação anterior nos permite caracterizar o uso de diversas técnicas que permitem a redução dos componentes do CPI geral. A Figura 3.1 mostra as técnicas que examinaremos neste capítulo e no Apêndice H, além dos tópicos abordados no material introdutório do

Técnica	Reduz	Seção
Encaminhamento e bypassing	Stalls de hazard de dados em potencial	C.2
Delayed branches e escalonamento de desvio simples	Stalls de hazard de controle	C.2
Escalonamento básico de pipeline do compilador	Stalls de hazard de dados	C.2, C.3.2
Escalonamento dinâmico básico (scoreboarding)	Stalls de hazard de dados de dependências verdadeiras	C.7
Desdobramento de loop	Stalls de hazard de controle	3.2
Previsão de desvio	Stalls de controle	3.3
Escalonamento dinâmico com renomeação	Stalls de hazard de dados e stalls de antidependências e dependências de saída	3.4
Especulação de hardware	Stalls de hazard de dados e hazard de controle	3.6
Desambiguidade dinâmica da memória	Stalls de hazard de dados com memória	3.6
Múltiplos despachos de instruções por ciclo	CPI ideal	3.7, 3.8
Análise de dependência do compilador, pipelining de software, trace scheduling	CPI ideal, stalls de hazard de dados	H.2, H.3
Suporte do hardware para especulação do compilador	CPI ideal, stalls de hazard de dados, stalls de hazard de desvios	H.4, H.5

FIGURA 3.1 As principais técnicas examinadas no Apêndice C, no Capítulo 3 ou no Apêndice H aparecem com o componente da equação do CPI afetado pela técnica.

Apêndice C. Neste capítulo, veremos que as técnicas introduzidas para diminuir o CPI de pipeline ideal podem aumentar a importância de lidar com os hazards.

O que é paralelismo em nível de instrução?

Todas as técnicas deste capítulo exploram o paralelismo entre as instruções. A quantidade de paralelismo disponível dentro de um *bloco básico* — uma sequência de código em linha reta, sem desvios para dentro, exceto na entrada, e sem desvios para fora, exceto na saída — é muito pequena. Para os programas MIPS típicos, a frequência média de desvio dinâmico normalmente fica entre 15-25%, significando que 3-6 instruções são executadas entre um par de desvios. Como essas instruções provavelmente dependem umas das outras, a quantidade de sobreposição que podemos explorar dentro de um bloco básico provavelmente será menor que o tamanho médio desse bloco. Para obter melhorias de desempenho substanciais, temos que explorar o ILP entre os diversos blocos básicos.

A maneira mais simples e mais comum de aumentar o ILP é explorar o paralelismo entre iterações de um loop. Esse tipo de paralelismo normalmente é chamado *paralelismo em nível de loop*. A seguir damos um exemplo simples de loop, que soma dois arrays de 1.000 elementos e é completamente paralelo:

```
para (i=1; i<=999; i=i+1)
       x[i] = x[i] + y[i];
```

Cada iteração do loop pode sobrepor qualquer outra iteração, embora dentro de cada uma delas exista pouca ou nenhuma oportunidade para sobreposição.

Existem diversas técnicas que examinaremos para converter esse paralelismo em nível de loop em paralelismo em nível de instrução. Basicamente, essas técnicas funcionam desdobrando o loop estaticamente pelo compilador (como na seção seguinte) ou dinamicamente pelo hardware (como nas Seções 3.5 e 3.6).

Um método alternativo importante para explorar o paralelismo em nível de loop é o uso de SIMD tanto em processadores vetoriais quanto em unidades de processamento gráfico

(GPUs), ambos abordados no Capítulo 4. Uma instrução SIMD explora o paralelismo em nível de dados, operando sobre um número pequeno a moderado de itens de dados em paralelo (geralmente 2-8). Uma instrução vetorial explora o paralelismo em nível de dados, operando sobre itens de dados em paralelo. Por exemplo, a sequência de código anterior, que de forma simples requer sete instruções por iteração (dois loads, um ass, um store, dois adress updates e um brach) para um total de 7.000 instruções, poderia ser executada com um quarto das instruções em algumas arquiteturas SIMD, onde quatro itens de dados são processados por instrução. Em alguns processadores vetoriais, a sequência poderia usar somente quatro instruções: duas instruções para carregar os vetores *x* e *y* da memória, uma instrução para somar os dois vetores e uma instrução para armazenar o vetor de resultado. Naturalmente, essas instruções seriam canalizadas em um pipeline e teriam latências relativamente longas, mas essas latências podem ser sobrepostas.

Dependências de dados e hazards

Determinar como uma instrução depende de outra é fundamental para determinar quanto paralelismo existe em um programa e como esse paralelismo pode ser explorado. Particularmente, para explorar o paralelismo em nível de instrução, temos de determinar quais instruções podem ser executadas em paralelo. Se duas instruções são *paralelas*, elas podem ser executadas simultaneamente em um pipeline de qualquer profundidade sem causar quaisquer stalls, supondo que o pipeline tenha recursos suficientes (logo, não existem hazards estruturais). Se duas instruções forem dependentes, elas não serão paralelas e precisam ser executadas em ordem, embora normalmente possam ser parcialmente sobrepostas. O segredo, nos dois casos, é determinar se uma instrução é dependente de outra.

Dependências de dados

Existem três tipos diferentes de dependência: *dependências de dados* (também chamadas dependências de dados verdadeiras), *dependências de nome* e *dependências de controle*. Uma instrução *j* é dependente de dados da instrução *i* se um dos seguintes for verdadeiro:

- a instrução *i* produz um resultado que pode ser usado pela instrução *j*; ou
- a instrução *j* é dependente de dados da instrução *k*, e a instrução *k* é dependente de dados da instrução *i*.

A segunda condição afirma simplesmente que uma instrução é dependente de outra se houver uma cadeia de dependências do primeiro tipo entre as duas instruções. Essa cadeia de dependência pode ter o tamanho do programa inteiro. Observe que uma dependência dentro de uma única instrução (como ADDD R1,R1,R1) não é considerada dependência.

Por exemplo, considere a sequência de código MIPS a seguir, que incrementa um vetor de valores na memória (começando com 0(R1), e com o último elemento em 8(R2)), por um escalar no registrador F2 (para simplificar, no decorrer deste capítulo nossos exemplos ignoram os efeitos dos delayed branches).

```
Loop:    L.D    F0,0(R1)    ;F0=elemento do array
         ADD.D  F4,F0,F2    ;soma escalar em F2
         S.D    F4,0(R1)    ;armazena resultado
         DADDUI R1,R1,#-8   ;decrementa ponteiro em 8 bytes
         BNE    R1,R2,LOOP  ;desvia R1!=R2
```

As dependências de dados nessa sequência de código envolvem tanto dados de ponto flutuante

```
Loop:    L.D    F0,0(R1)    ;F0=elemento do array

         ADD.D  F4,F0,F2    ;soma escalar em F2

         S.D    F4,0(R1)    ;armazena resultado
```

quanto dados inteiros:

```
DADDIU  R1,R1,#-8  ;decrement pointer
        |          ;8 bytes (per DW)
        v
 BNE    R1,R2,Loop ;branch R1!=R2
```

As duas sequências dependentes anteriores, conforme mostrado pelas setas, têm cada instrução dependendo da anterior. As setas aqui e nos exemplos seguintes mostram a ordem que deve ser preservada para a execução correta. A seta sai de uma instrução que deve preceder a instrução para a qual ela aponta.

Se duas instruções forem dependentes de dados, elas não poderão ser executadas simultaneamente nem ser completamente sobrepostas. A dependência implica que haverá uma cadeia de um ou mais hazards de dados entre as duas instruções (ver no Apêndice C uma rápida descrição dos hazards de dados, que definiremos com exatidão mais adiante). A execução simultânea das instruções fará um processador com pipeline interlock (e uma profundidade de pipeline maior que a distância entre as instruções em ciclos) detectar um hazard e parar (stall), reduzindo ou eliminando assim a sobreposição. Em um processador sem interlock, que conta com o escalonamento do compilador, o compilador não pode escalonar instruções dependentes de modo que elas sejam totalmente sobrepostas, pois o programa não será executado corretamente. A presença de dependência de dados em uma sequência de instruções reflete uma dependência de dados no código-fonte a partir do qual a sequência de instruções foi gerada. O efeito da dependência de dados original precisa ser preservado.

As dependências são uma propriedade dos *programas*. Se determinada dependência resulta em um hazard real sendo detectado e se esse hazard realmente causa um stall, essas são propriedades da *organização do pipeline*. Essa diferença é essencial para entender como o paralelismo em nível de instrução pode ser explorado.

Uma dependência de dados transmite três coisas: 1) a possibilidade de um hazard; 2) a ordem em que os resultados podem ser calculados; e 3) um limite máximo de paralelismo a ser explorado. Esses limites serão explorados em detalhes na Seção 3.10 e no Apêndice H.

Como uma dependência de dados pode limitar a quantidade de paralelismo em nível de instrução que podemos explorar, um foco importante deste capítulo é contornar essas limitações. Uma dependência pode ser contornada de duas maneiras diferentes: mantendo a dependência, mas evitando o hazard, e eliminando uma dependência, transformando o código. O escalonamento do código é o método principal utilizado para evitar hazards sem alterar uma dependência, e esse escalonamento pode ser feito tanto pelo compilador quanto pelo hardware.

Um valor de dados pode fluir entre as instruções ou por registradores ou por locais da memória. Quando o fluxo de dados ocorre em um registrador, a detecção da dependência é direta, pois os nomes dos registradores são fixos nas instruções, embora ela fique mais complicada quando os desvios intervêm e questões de exatidão forçam o compilador ou o hardware a ser conservador.

As dependências que fluem pelos locais da memória são mais difíceis de se detectar, pois dois endereços podem referir-se ao mesmo local, mas podem aparecer de formas diferentes. Por exemplo, 100(R4) e 20(R6) podem ser endereços de memória idênticos. Além disso, o endereço efetivo de um load ou store pode mudar de uma execução da instrução para outra (de modo que 20(R4) e 20(R4) podem ser diferentes), complicando ainda mais a detecção de uma dependência.

Neste capítulo, examinaremos o hardware para detectar as dependências de dados que envolvem locais de memória, mas veremos que essas técnicas também possuem limitações.

As técnicas do compilador para detectar essas dependências são críticas para desvendar o paralelismo em nível de loop.

Dependências de nome

O segundo tipo de dependência é uma *dependência de nome*. Uma dependência de nome ocorre quando duas instruções usam o mesmo registrador ou local de memória, chamado *nome*, mas não existe fluxo de dados entre as instruções associadas a ele. Existem dois tipos de dependências de nome entre uma instrução *i* que *precede* uma instrução *j* na ordem do programa:

1. Uma *antidependência* entre a instrução *i* e a instrução *j* ocorre quando a instrução *j* escreve, em um registrador ou local de memória, que a instrução *i* lê. A ordenação original precisa ser preservada para garantir que *i* leia o valor correto. No exemplo das páginas 130 e 131, existe uma antidependência entre S.D e DADDIU no registrador R1.
2. Uma *dependência de saída* ocorre quando a instrução *i* e a instrução *j* escrevem no mesmo registrador ou local de memória. A ordenação entre as instruções precisa ser preservada para garantir que o valor finalmente escrito corresponda à instrução *j*.

As antidependências e as dependências de saída são dependências de nome, ao contrário das verdadeiras dependências de dados, pois não existe valor sendo transmitido entre as instruções. Como uma dependência de nome não é uma dependência verdadeira, as instruções envolvidas em uma dependência de nome podem ser executadas simultaneamente ou ser reordenadas se o nome (número de registrador ou local de memória) usado nas instruções for alterado de modo que as instruções não entrem em conflito.

Essa renomeação pode ser feita com mais facilidade para operandos registradores, quando é chamada *renomeação de registrador.* A renomeação de registrador pode ser feita estaticamente por um compilador ou dinamicamente pelo hardware. Antes de descrever as dependências que surgem dos desvios, vamos examinar o relacionamento entre as dependências e os hazards de dados do pipeline.

Hazards de dados

Um hazard é criado sempre que existe uma dependência entre instruções, e elas estão próximas o suficiente para que a sobreposição durante a execução mude a ordem de acesso ao operando envolvido na dependência. Devido à dependência, temos de preservar a chamada *ordem do programa*, ou seja, a ordem em que as instruções seriam executadas se executadas sequencialmente uma de cada vez, conforme determinado pelo programa original. O objetivo do nosso software e das técnicas de hardware é explorar o paralelismo, preservando a ordem do programa *somente onde afeta o resultado do programa*. A detecção e a prevenção dos hazards garantem a preservação da ordem necessária do programa.

Os hazard de dados, que são descritos informalmente no Apêndice C, podem ser classificados em um de três tipos, dependendo da ordem de acessos de leitura e escrita nas instruções. Por convenção, os hazards são nomeados pela ordenação no programa, que precisa ser preservada pelo pipeline. Considere duas instruções *i* e *j*, com *i* precedendo *j* na ordem do programa. Os hazards de dados possíveis são:

- RAW (*Read After Write* — leitura após escrita) — *j* tenta ler um fonte antes que *i* escreva nele, de modo que *j* apanha incorretamente o valor *antigo*. Esse hazard é o tipo mais comum e corresponde a uma dependência de dados verdadeira. A ordem do programa precisa ser preservada para garantir que *j* recebe o valor de *i*.
- WAW (*Write After Write* — escrita após escrita) — *j* tenta escrever um operando antes que ele seja escrito por *i*. As escritas acabam sendo realizadas na ordem errada, deixando o valor escrito por *i* em vez do valor escrito por *j* no destino. Esse hazard

corresponde a uma dependência de saída. Hazards WAW estão presentes apenas em pipelines que escrevem em mais de um estágio de pipe ou permitem que uma instrução prossiga mesmo quando uma instrução anterior é parada.

- WAR (*Write After Read* — escrita após leitura) — *j* tenta escrever um destino antes que seja lido por *i*, de modo que *i* incorretamente apanha o valor *novo*. Esse hazard surge de uma antidependência. Hazards WAR não podem ocorrer na maioria dos pipelines de despacho estático — até mesmo os pipelines mais profundos ou pipelines de ponto flutuante —, pois todas as leituras vêm cedo (em ID) e todas as escritas vêm tarde (em WB) (para se convencer, Apêndice A). Um hazard WAR ocorre quando existem algumas instruções que escrevem resultados cedo no pipeline de instruções *e* outras instruções que leem um fonte tarde no pipeline ou quando as instruções são reordenadas, como veremos neste capítulo.

Observe que o caso RAR (*Read After Read* — leitura após leitura) não é um hazard.

Dependências de controle

O último tipo de dependência é uma *dependência de controle*. Uma dependência de controle determina a ordenação de uma instrução *i* com relação a uma instrução de desvio, de modo que essa instrução seja executada na ordem correta do programa e somente quando precisar. Cada instrução, exceto aquelas no primeiro bloco básico do programa, é dependente de controle em algum conjunto de desvios e, em geral, essas dependências de controle precisam ser preservadas para preservar a ordem do programa. Um dos exemplos mais simples de uma dependência de controle é a dependência das instruções na parte "then" de uma instrução "if" no desvio. Por exemplo, no segmento de código

```
if p1 {
     S1;
};
if p2 {
     S2;
}
```

S1 é dependente de controle de p1, e S2 é dependente de controle de p2, mas não de p1.

Em geral, existem duas restrições impostas pelas dependências de controle:

1. Uma instrução que é dependente de controle em um desvio não pode ser movida *antes* do desvio, de modo que sua execução *não é mais controlada* por ele. Por exemplo, não podemos apanhar uma instrução da parte then de uma instrução if e movê-la para antes da instrução if.
2. Uma instrução que não é dependente de controle em um desvio não pode ser movida para *depois* do desvio, de modo que sua execução *é controlada* pelo desvio. Por exemplo, não podemos apanhar uma instrução antes da instrução if e movê-la para a parte then.

Quando os processadores preservam a ordem estrita do programa, eles garantem que as dependências de controle também sejam preservadas. Porém, podemos estar querendo executar instruções que não deveriam ter sido executadas, violando assim as dependências de controle, *se* pudermos fazer isso sem afetar a exatidão do programa. A dependência de controle não é a propriedade crítica que precisa ser preservada. Em vez disso, as duas propriedades críticas à exatidão do programa — e normalmente preservadas mantendo-se a dependência de dados e o controle — são o *comportamento de exceção* e o *fluxo de dados*.

A preservação do comportamento de exceção significa que quaisquer mudanças na ordem de execução da instrução não deverão mudar o modo como as exceções são geradas no programa. Normalmente, isso significa que a reordenação da execução da instrução não deverá causar quaisquer novas exceções no programa. Um exemplo simples mostra como a

manutenção das dependências de controle e dados pode impedir tais situações. Considere essa sequência de código:

```
DADDU    R2,R3,R4
BEQZ     R2,L1
LW       R1,0(R2)
L1:
```

Nesse caso, é fácil ver que, se não mantivermos a dependência de dados envolvendo R2, poderemos alterar o resultado do programa. Menos óbvio é o fato de que, se ignorarmos a dependência de controle e movermos as instruções load para antes do desvio, elas poderão causar uma exceção de proteção de memória. Observe que *nenhuma dependência de dados* nos impede de trocar o BEQZ e o LW; essa é apenas a dependência de controle. Para permitir que reordenemos essas instruções (e ainda preservemos a dependência de dados), gostaríamos apenas de ignorar a exceção quando o desvio for tomado. Na Seção 3.6, veremos uma técnica de hardware, a *especulação*, que nos permite contornar esse problema de exceção. O Apêndice H examina as técnicas de software para dar suporte à especulação.

A segunda propriedade preservada pela manutenção das dependências de dados e das dependências de controle é o fluxo de dados. O *fluxo de dados* é o fluxo real dos valores de dados entre as instruções que produzem resultados e aquelas que os consomem. Os desvios tornam o fluxo de dados dinâmico, pois permitem que a fonte de dados para determinada instrução venha de muitos pontos. Em outras palavras, é insuficiente apenas manter dependências de dados, pois uma instrução pode ser dependente de dados em mais de um predecessor. A ordem do programa é o que determina qual predecessor realmente entregará um valor de dados a uma instrução. A ordem do programa é garantida mantendo-se as dependências de controle.

Por exemplo, considere o seguinte fragmento de código:

```
DADDU      R1,R2,R3
BEQZ       R4,L
DSUBU      R1,R5,R6
L:         ...
OR         R7,R1,R8
```

Neste exemplo, o valor de R1 usado pela instrução OR depende de o desvio ser tomado ou não. A dependência de dados sozinha não é suficiente para preservar a exatidão. A instrução OR é dependente de dados nas instruções DADDU e DSUBU, mas somente preservar essa ordem é insuficiente para a execução correta.

Em vez disso, quando as instruções são executadas, o fluxo de dados precisa ser preservado: se o desvio não for tomado, o valor de R1 calculado pelo DSUBU deve ser usado pelo OR e, se o desvio for tomado, o valor de R1 calculado pelo DADDU deve ser usado pelo OR. Preservando a dependência de controle do OR no desvio, impedimos uma mudança ilegal no fluxo dos dados. Por motivos semelhantes, a instrução DSUBU não pode ser movida para cima do desvio. A especulação, que ajuda com o problema de exceção, também nos permite suavizar o impacto da dependência de controle enquanto ainda mantém o fluxo de dados, conforme veremos na Seção 3.6.

Às vezes, podemos determinar que a violação da dependência de controle não pode afetar o comportamento da exceção ou o fluxo de dados. Considere a sequência de código a seguir:

```
          DADDU      R1,R2,R3
          BEQZ       R12,skip
          DSUBU      R4,R5,R6
          DADDU      R5,R4,R9
skip:     OR         R7,R8,R9
```

Suponha que saibamos que o destino do registrador da instrução DSUBU (R4) não foi usado depois da instrução rotulada com skip (a propriedade que informa se um valor será usado por uma instrução vindoura é chamada de *liveness*). Se R4 não fosse utilizado, a mudança do valor de R4 imediatamente antes do desvio não afetaria o fluxo de dados, pois R4 estaria *morto* (em vez de vivo) na região do código após skip. Assim, se R4 estivesse morto e a instrução DSUBU existente não pudesse gerar uma exceção (outra além daquelas das quais o processador retoma o processo), poderíamos mover a instrução DSUBU para antes do desvio, pois o fluxo de dados não poderia ser afetado por essa mudança.

Se o desvio for tomado, a instrução DSUBU será executada e não terá utilidade, mas não afetará os resultados do programa. Esse tipo de escalonamento de código também é uma forma de especulação, normalmente chamada de especulação de software, pois o compilador está apostando no resultado do desvio; nesse caso, a aposta é que o desvio normalmente não é tomado. O Apêndice H discute mecanismos mais ambiciosos de especulação do compilador. Normalmente ficará claro, quando dissermos especulação ou especulativo, se o mecanismo é um mecanismo de hardware ou software; quando isso não for claro, é melhor dizer "especulação de hardware" ou "especulação de software".

A dependência de controle é preservada pela implementação da detecção de hazard de controlar um stall de controle. Stalls de controle podem ser eliminados ou reduzidos por diversas técnicas de hardware e software, que examinaremos na Seção 3.3.

3.2 TÉCNICAS BÁSICAS DE COMPILADOR PARA EXPOR O ILP

Esta seção examina o uso da tecnologia simples de compilação para melhorar a capacidade de um processador de explorar o ILP. Essas técnicas são cruciais para os processadores que usam despacho estático e escalonamento estático. Armados com essa tecnologia de compilação, examinaremos rapidamente o projeto e o desempenho de processadores usando despacho estático. O Apêndice H investigará esquemas mais sofisticados de compilação e hardware associado, projetados para permitir que um processador explore mais o paralelismo em nível de instrução.

Escalonamento básico de pipeline e desdobramento de loop

Para manter um pipeline cheio, o paralelismo entre as instruções precisa ser explorado encontrando-se sequências de instruções não relacionadas que possam ser sobrepostas no pipeline. Para evitar stall de pipeline, uma instrução dependente precisa ser separada da instrução de origem por uma distância em ciclos de clock igual à latência do pipeline dessa instrução de origem. A capacidade de um compilador de realizar esse escalonamento depende da quantidade de ILP disponível no programa e das latências das unidades funcionais no pipeline. A Figura 3.2 mostra as latências da unidade de PF que consideramos neste capítulo, a menos que latências diferentes sejam indicadas explicitamente. Consideramos o pipeline de inteiros-padrão de cinco estágios, de modo que os desvios possuem um atraso de um ciclo de clock. Consideramos que as unidades funcionais são totalmente pipelined ou replicadas (tantas vezes quanto for a profundidade do pipeline), de modo que uma operação de qualquer tipo possa ser enviada em cada ciclo de clock e não haja hazards estruturais.

Nesta subseção, examinaremos como o compilador pode aumentar a quantidade de ILP disponível transformando loops. Esse exemplo serve tanto para ilustrar uma técnica importante quanto para motivar as transformações de programa mais poderosas, descritas

Instrução produzindo resultado	Instrução usando resultado	Latência em ciclos de clock
Op. ALU de PF	Outra op. ALU de PF	3
Op. ALU de PF	Store duplo	2
Load duplo	Op. ALU de PF	1
Load duplo	Store duplo	0

FIGURA 3.2 Latências de operações de PF usadas neste capítulo.
A última coluna é o número de ciclos de clock interferindo, necessários para evitar um stall. Esses números são semelhantes às latências médias que veríamos em uma unidade de PF. A latência de um load de ponto flutuante para um store é 0, pois o resultado do load pode ser contornado sem protelar o store. Vamos continuar considerando uma latência de load de inteiros igual a 1 e uma latência de operação da ALU igual a 0.

no Apêndice H. Vamos nos basear no seguinte segmento de código, que acrescenta um valor escalar a um vetor:

```
para (i=999; i>=0; i=i-1)
       x[i] = x[i] + s;
```

Podemos ver que esse loop é paralelo, observando que o corpo de cada iteração é independente. Formalizaremos essa noção no Apêndice H, descrevendo como podemos testar se as iterações do loop são independentes no momento da compilação. Primeiro, vejamos o desempenho desse loop, mostrando como podemos usar o paralelismo para melhorar seu desempenho para um pipeline MIPS com as latências indicadas antes.

O primeiro passo é traduzir o segmento anterior para a linguagem assembly MIPS. No segmento de código a seguir, R1 é inicialmente o endereço do elemento no array com o endereço mais alto, e F2 contém o valor escalar *s*. O registrador R2 é pré-calculado, de modo que 8(R2) é o endereço do último elemento a ser processado.

O código MIPS direto, não escalonado para o pipeline, se parece com este:

```
Loop:   L.D      F0,0(R1)     ;F0=array element
        ADD.D    F4,F0,F2     ;add scalar in F2
        S.D      F4,0(R1)     ;store result
        DADDUI   R1,R1,#-8    ;decrement pointer
                              ;8 bytes (per DW)
        BNE      R1,R2,Loop   ;branch R1!=R2
```

Vamos começar vendo como esse loop funcionará quando programado em um pipeline simples para MIPS com as latências da Figura 3.2.

Exemplo Mostre como o loop ficaria no MIPS, escalonado e não escalonado, incluindo quaisquer stalls ou ciclos de clock ociosos. Escalone para os atrasos das operações de ponto flutuante, mas lembre-se de que estamos ignorando os delayed branches.

Resposta Sem qualquer escalonamento, o loop será executado da seguinte forma, usando nove ciclos:

```
                                        Ciclo de clock emitido
Loop:   L.D      F0,0(R1)                1
        stall                            2
        ADD.D    F4,F0,F2                3
        stall                            4
        stall                            5
        S.D      F4,0(R1)                6
        DADDUI   R1,R1,#-8               7
        stall                            8
        BNE      R1,R2,Loop              9
```

Podemos escalonar o loop para obter apenas dois stalls e reduzir o tempo para sete ciclos:

```
Loop:  L.D     F0,0(R1)
       DADDUI  R1,R1,#-8
       ADD.D   F4,F0,F2
       stall
       stall
       S.D     F4,8(R1)
       BNE     R1,R2,Loop
```

Os stalls após ADD.D são para uso do S.D.

No exemplo anterior, completamos uma iteração de loop e armazenamos um elemento do array a cada sete ciclos de clock, mas o trabalho real de operar sobre o elemento do array leva apenas três (load, add e store) desses sete ciclos de clock. Os quatro ciclos de clock restantes consistem em overhead do loop — o DADDUI e o BNE — e dois stalls. Para eliminar esses quatro ciclos de clock, precisamos apanhar mais operações relativas ao número de instruções de overhead.

Um esquema simples para aumentar o número de instruções relativas às instruções de desvio e overhead é o *desdobramento de loop*. O desdobramento simplesmente replica o corpo do loop várias vezes, ajustando o código de término do loop.

O desdobramento de loop também pode ser usado para melhorar o escalonamento. Por eliminar o desvio, ele permite que instruções de diferentes iterações sejam escalonadas juntas. Nesse caso, podemos eliminar os stalls de uso de dados criando instruções independentes adicionais dentro do corpo do loop. Se simplesmente replicássemos as instruções quando desdobrássemos o loop, o uso resultante dos mesmos registradores poderia nos impedir de escalonar o loop com eficiência. Assim, desejaremos usar diferentes registradores para cada iteração, aumentando o número de registradores exigidos.

Exemplo Mostre nosso loop desdobrado de modo que haja quatro cópias do corpo do loop, considerando que R1 – R2 (ou seja, o tamanho do array) é inicialmente um múltiplo de 32, o que significa que o número de iterações do loop é um múltiplo de 4. Elimine quaisquer cálculos obviamente redundantes e não reutilize qualquer um dos registradores.

Resposta Aqui está o resultado depois de mesclar as instruções DADDUI e remover as operações BNE desnecessárias que são duplicadas durante o desdobramento. Observe que agora R2 precisa ser definido de modo que 32(R2) seja o endereço inicial dos quatro últimos elementos.

```
Loop:  L.D      F0,0(R1)
       ADD.D    F4,F0,F2
       S.D      F4,0(R1)      ;drop DADDUI & BNE
       L.D      F6,-8(R1)
       ADD.D    F8,F6,F2
       S.D      F8,-8(R1)     ;drop DADDUI & BNE
       L.D      F10,-16(R1)
       ADD.D    F12,F10,F2
       S.D      F12,-16(R1)   ;drop DADDUI & BNE
       L.D      F14,-24(R1)
       ADD.D    F16,F14,F2
       S.D      F16,-24(R1)
       DADDUI   R1,R1,#-32
       BNE      R1,R2,Loop
```

Eliminamos três desvios e três decrementos de R1. Os endereços nos loads e stores foram compensados para permitir que as instruções DADDUI em R1 sejam mescladas. Essa otimização pode parecer trivial, mas não é; ela exige substituição simbólica e simplificação. A substituição simbólica e a simplificação rearrumarão expressões de modo a permitir que constantes sejam reduzidas, possibilitando que uma expressão como "((i + 1) + 1)" seja reescrita como "(i + (1 + 1))" e depois simplificada para "(i + 2)". Veremos as formas mais gerais dessas otimizações que eliminam cálculos dependentes no Apêndice H.

Sem o escalonamento, cada operação no loop desdobrado é seguida por uma operação dependente e, assim, causará um stall. Esse loop será executado em 27 ciclos de clock — cada LD tem um stall, cada ADDD tem dois, o DADDUI tem um mais 14 ciclos de despacho de instrução — ou 6,75 ciclos de clock para cada um dos quatro elementos, mas ele pode ser escalonado para melhorar significativamente o desempenho. O desdobramento do loop normalmente é feito antes do processo de compilação, de modo que cálculos redundantes podem ser expostos e eliminados pelo otimizador.

Em programas reais, normalmente não sabemos o limite superior no loop. Suponha que ele seja n e que gostaríamos de desdobrar o loop para criar k cópias do corpo. Em vez de um único loop desdobrado, geramos um par de loops consecutivos. O primeiro executa (n mod k) vezes e tem um corpo que é o loop original. O segundo é o corpo desdobrado, cercado por um loop externo que repete (n/k) vezes (como veremos no Capítulo 4, essa técnica é similar a uma técnica chamada *strip mining*, usada em compiladores para processadores vetoriais). Para valores grandes de n, a maior parte do tempo de execução será gasta no corpo do loop desdobrado.

No exemplo anterior, o desdobramento melhora o desempenho desse loop, eliminando as instruções de overhead, embora aumente o tamanho do código substancialmente. Como o loop desdobrado funcionará quando for escalonado para o pipeline descrito anteriormente?

Exemplo Mostre o loop desdobrado no exemplo anterior após ter sido escalonado para o pipeline com as latências mostradas na Figura 3.2.

Resposta

```
Loop:  L.D      F0,0(R1)
       L.D      F6,-8(R1)
       L.D      F10,-16(R1)
       L.D      F14,-24(R1)
       ADD.D    F4,F0,F2
       ADD.D    F8,F6,F2
       ADD.D    F12,F10,F2
       ADD.D    F16,F14,F2
       S.D      F4,0(R1)
       S.D      F8,-8(R1)
       DADDUI   R1,R1,#-32
       S.D      F12,16(R1)
       S.D      F16,8(R1)
       BNE      R1,R2,Loop
```

O tempo de execução do loop caiu para um total de 14 ciclos de clock ou 3,5 ciclos de clock por elemento, em comparação com os nove ciclos por elemento antes de qualquer desdobramento ou escalonamento e sete ciclos quando escalonado, mas não desdobrado.

O ganho vindo do escalonamento no loop desdobrado é ainda maior que no loop original. Esse aumento surge porque o desdobramento do loop expõe mais computação que pode ser escalonada para minimizar os stalls; o código anterior não possui stalls. Dessa forma, o

escalonamento do loop necessita da observação de que os loads e stores são independentes e podem ser trocados.

Resumo do desdobramento e escalonamento de loop

No decorrer deste capítulo e no Apêndice H, veremos uma série de técnicas de hardware e software que nos permitirão tirar proveito do paralelismo em nível de instrução para utilizar totalmente o potencial das unidades funcionais em um processador. A chave para a maioria dessas técnicas é saber quando e como a ordenação entre as instruções pode ser alterada. No nosso exemplo, fizemos muitas dessas mudanças, que, para nós, como seres humanos, eram obviamente permissíveis. Na prática, esse processo precisa ser realizado em um padrão metódico, seja por um compilador, seja pelo hardware. Para obter o código desdobrado final, tivemos de tomar as seguintes decisões e transformações:

- Determinar que o desdobramento do loop seria útil descobrindo que as iterações do loop eram independentes, exceto para o código de manutenção do loop.
- Usar diferentes registradores para evitar restrições desnecessárias que seriam forçadas pelo uso dos mesmos registradores para diferentes cálculos.
- Eliminar as instruções extras de teste e desvio, e ajustar o código de término e iteração do loop.
- Determinar que os loads e stores no loop desdobrado podem ser trocados, observando que os loads e stores de diferentes iterações são independentes. Essa transformação requer analisar os endereços de memória e descobrir que eles não se referem ao mesmo endereço.
- Escalonar o código preservando quaisquer dependências necessárias para gerar o mesmo resultado do código original.

O requisito-chave por trás de todas essas transformações é o conhecimento de como uma instrução depende de outra e como as instruções podem ser alteradas ou reordenadas dadas as dependências.

Existem três tipos de limite diferentes para os ganhos que podem ser alcançados pelo desdobramento do loop: 1) diminuição na quantidade de overhead amortizado com cada desdobramento; 2) limitações de tamanho de código e 3) limitações do compilador. Vamos considerar primeiro a questão do overhead do loop. Quando desdobramos o loop quatro vezes, ele gerou paralelismo suficiente entre as instruções em que o loop poderia ser escalonado sem ciclos de stall. De fato, em 14 ciclos de clock, somente dois ciclos foram overhead do loop: o DADDUI, que mantém o valor de índice, e o BNE, que termina o loop. Se o loop for desdobrado oito vezes, o overhead será reduzido de 1/2 ciclo por iteração original para 1/4.

Um segundo limite para o desdobramento é o consequente crescimento no tamanho do código. Para loops maiores, o crescimento no tamanho do código pode ser um problema, particularmente se causar aumento na taxa de falha da cache de instruções.

Outro fator normalmente mais importante que o tamanho do código é o déficit em potencial de registradores, que é criado pelo desdobramento e pelo escalonamento agressivo. Esse efeito secundário que resulta do escalonamento de instruções em segmentos de código grandes é chamado *pressão de registradores.* Ele surge porque escalonar o código para aumentar o ILP faz com que o número de valores vivos seja aumentado. Talvez depois do escalonamento de instrução agressivo não seja possível alocar todos os valores vivos aos registradores. O código transformado, embora teoricamente mais rápido, pode perder parte de sua vantagem ou toda ela, pois gera uma escassez de registradores. Sem desdobramento, o escalonamento agressivo é suficientemente limitado pelos desvios, de

modo que a pressão de registradores raramente é um problema. Entretanto, a combinação de desdobramento e escalonamento agressivo pode causar esse problema. O problema se torna especialmente desafiador nos processadores de múltiplo despacho, que exigem a exposição de mais sequências de instruções independentes, cuja execução pode ser sobreposta. Em geral, o uso de transformações de alto nível sofisticadas, cujas melhorias em potencial são difíceis de medir antes da geração de código detalhada, levou a aumentos significativos na complexidade dos compiladores modernos.

O desdobramento de loop é um método simples, porém útil, para aumentar o tamanho dos fragmentos de código direto que podem ser escalonados com eficiência. Essa transformação é útil em diversos processadores, desde pipelines simples, como aqueles que examinamos até aqui, até os superescalares e VLIWs de múltiplo despacho, explorados mais adiante neste capítulo.

3.3 REDUÇÃO DE CUSTOS COM PREVISÃO DE DESVIO AVANÇADO

Devido à necessidade de forçar as dependências de controle por meio dos hazards de dados e stalls, os desvios atrapalharão o desempenho do pipeline. O desdobramento de loop é uma forma de reduzir o número de hazards de desvio; também podemos reduzir as perdas de desempenho prevendo como elas se comportarão. O comportamento dos desvios pode ser previsto estaticamente no momento da compilação e dinamicamente pelo hardware no momento da execução. As previsões de desvio estático às vezes são usadas nos processadores em que a expectativa é de que o comportamento do desvio seja altamente previsível no momento da compilação; a previsão estática também pode ser usada para auxiliar na previsão dinâmica.

Correlacionando esquemas de previsão de desvio

Os esquemas de previsão de 2 bits utilizam apenas o comportamento recente de um único desvio para prever o comportamento futuro desse desvio. Talvez seja possível melhorar a exatidão da previsão se também virmos o comportamento recente dos *outros* desvios em vez de vermos apenas o desvio que estamos tentando prever. Considere um pequeno fragmento de código, do benchmark eqntott, um membro dos primeiros pacotes de benchmark SPEC que exibiam comportamento de previsão de desvio particularmente ruins:

```
if (aa==2)
      aa=0;
if (bb==2)
      bb=0;
if (aa!=bb) {
```

Aqui está o código MIPS que normalmente geraríamos para esse fragmento de código, considerando que aa e bb são atribuídos aos registradores R1 e R2:

```
       DADDIU   R3,R1,#-2
       BNEZ     R3,L1          ;branch b1   (aa!=2)
       DADD     R1,R0,R0       ;aa=0
L1:    DADDIU   R3,R2,#-2
       BNEZ     R3,L2          ;branch b2   (bb!=2)
       DADD     R2,R0,R0       ;bb=0
L2:    DSUBU    R3,R1,R2       ;R3=aa-bb
       BEQZ     R3,L3          ;branch b3   (aa==bb)
```

Vamos rotular esses desvios como b1, b2 e b3. A principal observação é que o comportamento do desvio b3 é correlacionado com o comportamento dos desvios b1 e b2. Obviamente, se os desvios b1 e b2 não forem tomados (ou seja, se as condições forem avaliadas

como verdadeira e aa e bb receberem o valor 0), então b3 será tomado, pois aa e bb são nitidamente iguais. Um esquema de previsão que utiliza o comportamento de um único desvio para prever o resultado desse desvio nunca poderá capturar esse comportamento.

Os esquemas de previsão de desvio que usam o comportamento de outros desvios para fazer uma previsão são chamados *previsores de correlação* ou *previsores de dois níveis*. Os previsores de correlação existentes acrescentam informações sobre o comportamento da maioria dos desvios recentes para decidir como prever determinado desvio. Por exemplo, um esquema de previsão (1,2) utiliza o comportamento do último desvio para escolher dentre um par de previsores de desvio de 2 bits na previsão de determinado desvio. No caso geral, um esquema de previsão (m,n) utiliza o comportamento dos últimos m desvios para escolher dentre 2^m previsores de desvio, cada qual sendo um previsor de n bits para um único desvio. A atração desse tipo de previsor de desvio de correlação é que ele pode gerar taxas de previsão mais altas do que o esquema de 2 bits e exige apenas uma quantidade trivial de hardware adicional.

A simplicidade do hardware vem de uma observação simples: a história global dos m desvios mais recentes pode ser registrada em um registrador de desvio de m bits, onde cada bit registra se o desvio foi tomado ou não. O buffer de previsão de desvio pode, então, ser indexado usando uma concatenação dos bits de baixa ordem a partir do endereço de desvio com um histórico global de m bits. Por exemplo, em um buffer (2,2) com 64 entradas no total, os 4 bits de endereço de baixa ordem do desvio (endereço de palavra) e os 2 bits globais representando o comportamento dos dois desvios executados mais recentemente formam um índice de 6 bits que pode ser usado para indexar os 64 contadores.

Quão melhor os previsores de desvio de correlação funcionam quando comparados com o esquema-padrão de 2 bits? Para compará-los de forma justa, temos de comparar os previsores que utilizam o mesmo número de bits de status. O número de bits em um previsor de (m,n) é

$$2^m \times n \times \text{Número de entradas de previsão selecionadas pelo endereço de desvio}$$

Um esquema de previsão de 2 bits sem histórico global é simplesmente um previsor (0,2).

Exemplo Quantos bits existem no previsor de desvio (0,2) com 4 K entradas? Quantas entradas existem em um previsor (2,2) com o mesmo número de bits?

Resposta O previsor com 4 K entradas possui

$$2^0 \times 2 \times 4\,\text{K} = 8\,\text{K bits}$$

Quantas entradas selecionadas de desvio existem em um previsor (2,2) que tem um total de 8 K bits no buffer de previsão? Sabemos que

$$2^2 \times 2 \times \text{Número de entradas de previsão selecionadas pelo desvio} = 8\,\text{K}$$

Logo, o número de entradas de previsão selecionadas pelo desvio = 1 K.

A Figura 3.3 compara as taxas de erro de previsão do previsor anterior (0,2) com 4 K entradas e um previsor (2,2) com 1 K entrada. Como você pode ver, esse previsor de correlação não apenas ultrapassa o desempenho de um previsor simples de 2 bits com o mesmo número total de bits de status, mas normalmente é superior a um previsor de 2 bits com um número ilimitado de entradas.

Previsores de torneio: combinando previsores locais e globais adaptativamente

A principal motivação para correlacionar previsores de desvio veio da observação de que o previsor de 2 bits padrão usando apenas informações locais falhou em alguns desvios

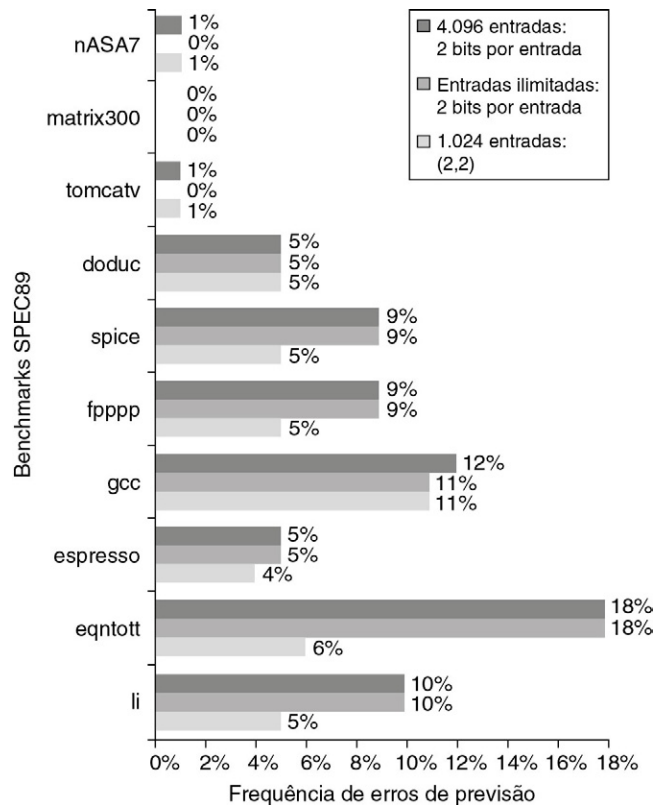



FIGURA 3.3 **Comparação de previsores de 2 bits.**
Primeiro previsor não correlacionado para 4.096 bits, seguido por um previsor não correlacionado de 2 bits com entradas ilimitadas e um previsor de 2 bits com 2 bits de histórico global e um total de 1.024 entradas. Embora esses dados sejam para uma versão mais antiga de SPEC, dados para benchmarks SPEC mais recentes mostrariam diferenças similares em precisão.

importantes e que, acrescentando informações globais, esse desempenho poderia ser melhorado. Os *previsores de torneio* levam essa compreensão para o próximo nível, usando vários previsores, normalmente um baseado em informações globais e outro em informações locais, e combinando-os com um seletor. Os previsores de torneio podem conseguir melhor exatidão em tamanhos médios (8-32 K bits) e também utilizar números muito grandes de bits de previsão com eficiência. Os previsores de torneio existentes utilizam um contador de saturação de 2 bits por desvio para escolher entre dois previsores diferentes com base em qual previsor (local, global ou até mesmo alguma mistura) foi mais eficaz nas previsões recentes. Assim como no previsor de 2 bits simples, o contador de saturação requer dois erros de previsão antes de alterar a identidade do previsor preferido.

A vantagem de um previsor de torneio é a sua capacidade de selecionar o previsor certo para determinado desvio, o que é particularmente crucial para os benchmarks de inteiros. Um previsor de torneio típico selecionará o previsor global em quase 40% do tempo para os benchmarks de inteiros SPEC e em menos de 15% do tempo para os benchmarks de PF SPEC. Além dos processadores Alpha, que foram pioneiros dos previsores de torneio, processadores AMD recentes, incluindo o Opteron e o Phenom, vêm usando previsores no estilo previsor de torneio.

A Figura 3.4 examina o desempenho de três previsores diferentes (um previsor local de 2 bits, um de correlação e um de torneio) para diferentes quantidades de bits usando o SPEC89 como benchmark. Como vimos anteriormente, a capacidade de previsão do

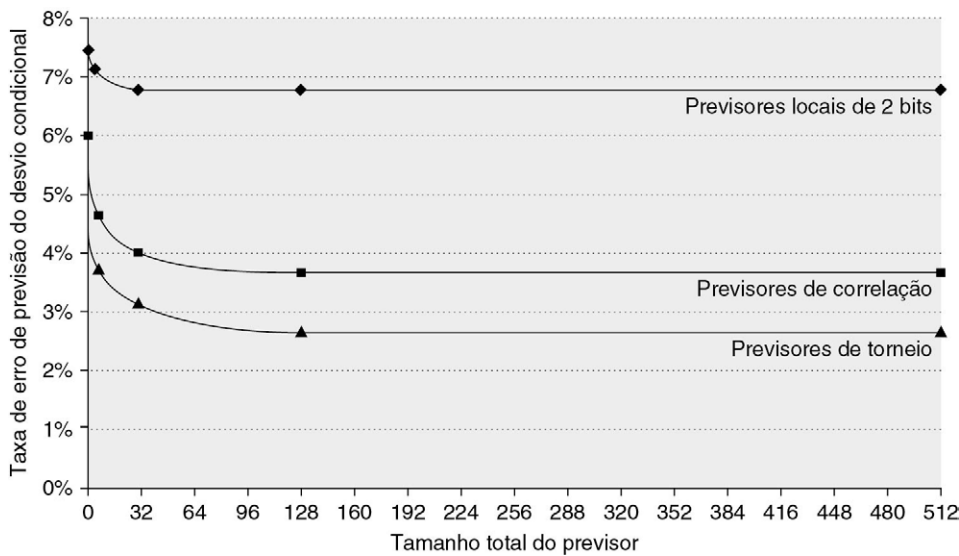



FIGURA 3.4 **Taxa de erro de previsão para três previsores diferentes no SPEC89 à medida que o número total de bits é aumentado.**
Os previsores são um previsor de 2 bits local, um previsor de correlação, que é idealmente estruturado em seu uso de informações globais e locais em cada ponto no gráfico, e um previsor de torneio. Embora esses dados sejam para uma versão mais antiga do SPEC, os dados para benchmarks SPEC mais recentes mostrariam comportamento semelhante, talvez convergindo para o limite assintótico em tamanhos de previsores ligeiramente maiores.

previsor local não melhora além de certo tamanho. O previsor de correlação mostra uma melhoria significativa, e o previsor de torneio gera um desempenho ligeiramente melhor. Para versões mais recentes do SPEC, os resultados seriam semelhantes, mas o comportamento assintomático não seria alcançado até que houvesse previsores de tamanho ligeiramente maior.

O previsor local consiste em um previsor de dois níveis. O nível superior é uma tabela de histórico consistindo em 1.024 entradas de 10 bits; cada entrada de 10 bits corresponde aos 10 resultados de desvio mais recentes para a entrada. Ou seja, se o desvio foi tomado 10 ou mais vezes seguidas em uma linha, a entrada na tabela local de histórico, serão todos 1s. Se o desvio for alternadamente tomado e não tomado, a entrada no histórico consistirá em 0s e 1s alternados. Esse histórico de 10 bits permite que padrões de até 10 desvios sejam descobertos e previstos. A entrada selecionada da tabela de histórico local é usada para indexar uma tabela de 1 K entradas consistindo em contadores de saturação de 3 bits, que oferecem a previsão local. Essa combinação, que usa um total de 29 K bits, leva a alta precisão na previsão do desvio.

O previsor de desvio Intel Core i7

A Intel liberou somente informações limitadas sobre o previsor de desvio do Core i7, que se baseia em previsores anteriores usados no chip Core Duo. O i7 usa um previsor de dois níveis que tem um previsor menor de primeiro nível, projetado para atender às restrições de ciclo de previsão de um desvio a cada ciclo de clock, e um previsor maior de segundo nível como backup. Cada previsor combina três previsores diferentes: 1) um previsor simples de dois bits, que foi apresentado no Apêndice C (e usado no previsor de torneio discutido anteriormente); 2) um previsor de histórico global, como aqueles que acabamos de ver; e 3) um previsor de saída de loop. O previsor de saída de loop usa um contador para prever o número exato de desvios tomados (que é o número de iterações de loop) para um desvio que é detectado como um desvio de loop. Para cada

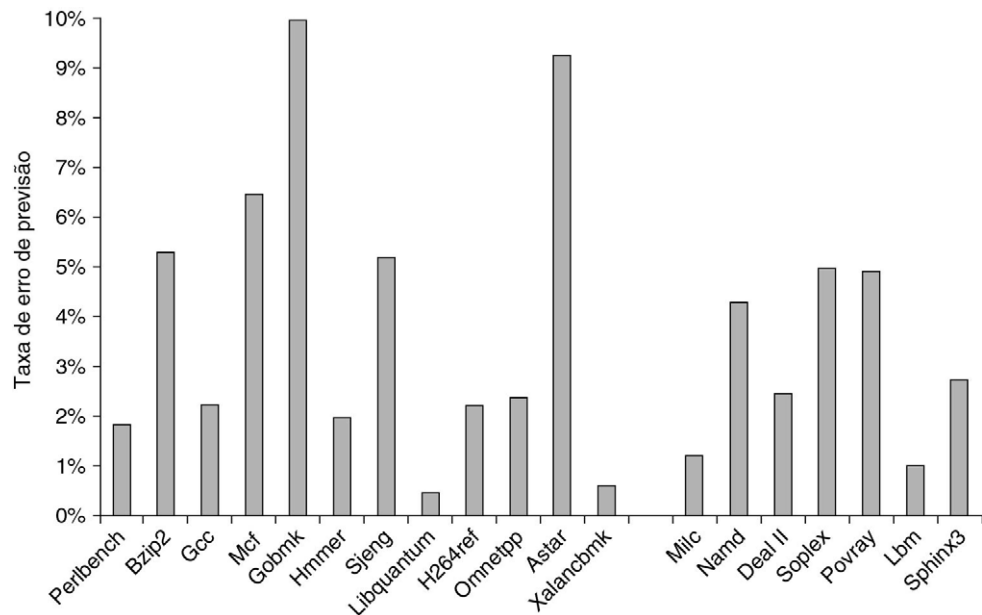



FIGURA 3.5 A taxa de erro de previsão para 19 dos benchmarks SPEC CPU2006 em comparação com o número de desvios removidos com sucesso em média para os benchmarks inteiros para PF (4% *versus* 3%). O mais importante é que ela é muito mais alta para poucos benchmarks.

desvio, a melhor previsão é selecionada entre os três previsores rastreando a precisão de cada previsão, como um previsor de torneio. Além desse previsor multinível principal, uma unidade separada prevê endereços-alvo para desvios indiretos e é usada também uma pilha para prever endereços de retorno.

Como em outros casos, a especulação cria alguns desafios na avaliação do previsor, uma vez que um desvio previsto de modo incorreto pode facilmente levar a busca e interpretação incorretas de outro desvio. Para manter a simplicidade, examinamos o número de previsões incorretas como uma porcentagem do número de desvios completados com sucesso (aqueles que não foram resultado de especulação incorreta). A Figura 3.5 mostra esses dados para 19 dos benchmarks SPEC CPU2006. Esses benchmarks são consideravelmente maiores do que o SPEC89 ou o SPEC2000, implicando que as taxas de previsão incorreta sejam ligeiramente maiores do que aquelas na Figura 3.4, mesmo com uma combinação mais elaborada de previsores. Uma vez que a previsão incorreta de desvios leva à especulação ineficaz, ela contribui para o trabalho perdido, como veremos mais adiante neste capítulo.

3.4 CONTORNANDO HAZARDS DE DADOS COM O ESCALONAMENTO DINÂMICO

Um pipeline simples escalonado estaticamente carrega uma instrução e a envia, a menos que haja uma dependência de dados entre uma instrução já no pipeline e a instrução carregada, que não pode ser escondida com o bypassing ou o encaminhamento (a lógica de encaminhamento reduz a latência efetiva do pipeline, de modo que certas dependências não resultam em hazards). Se houver uma dependência de dados que não possa ser escondida, o hardware de detecção de hazard forçará um stall no pipeline, começando com a instrução que usa o resultado. Nenhuma instrução nova é carregada ou enviada até que a dependência seja resolvida.

Nesta seção, exploramos o *escalonamento dinâmico*, em que o hardware reorganiza a execução da instrução para reduzir os stalls enquanto mantém o fluxo de dados e o comportamento da exceção. O escalonamento dinâmico oferece diversas vantagens: ele permite o tratamento de alguns casos quando as dependências são desconhecidas durante a compilação (p. ex., podem envolver uma referência à memória) e simplifica o compilador. E, talvez, o mais importante: ele permite que o processador tolere atrasos imprevistos, como falhas de cache, executando outro código enquanto espera que a falha seja resolvida. Quase tão importante, o escalonamento dinâmico permite que o código compilado com um pipeline em mente seja executado de forma eficiente em um pipeline diferente. Na Seção 3.6, exploraremos a especulação de hardware, uma técnica com vantagens significativas no desempenho, que é baseada no escalonamento dinâmico. Conforme veremos, as vantagens do escalonamento dinâmico são obtidas à custa de um aumento significativo na complexidade do hardware.

Embora um processador dinamicamente escalonado não possa mudar o fluxo de dados, ele tenta evitar os stalls quando as dependências estão presentes. Ao contrário, o escalonamento estático do pipeline pelo compilador (explicado na Seção 3.2) tenta minimizar os stalls separando instruções dependentes de modo que não levem a hazards. Naturalmente, o escalonamento de pipeline do compilador também pode ser usado no código destinado a executar em um processador com um pipeline escalonado dinamicamente.

Escalonamento dinâmico: a ideia

Uma limitação importante das técnicas de pipelining simples é que elas utilizam o despacho e a execução de instruções em ordem: as instruções são enviadas na ordem do programa e, se uma instrução for protelada no pipeline, nenhuma instrução posterior poderá prosseguir. Assim, se houver uma dependência entre duas instruções próximas no pipeline, isso levará a um hazard e ocorrerá um stall. Se houver várias unidades funcionais, essas unidades poderão ficar ociosas. Se a instrução *j* depender de uma instrução de longa execução *i*, em execução no pipeline, então todas as instruções depois de *j* precisarão ser proteladas até que *i* termine e *j* possa ser executada. Por exemplo, considere este código:

```
DIV.D     F0,F2,F4
ADD.D     F10,F0,F8
SUB.D     F12,F8,F14
```

A instrução SUB.D não pode ser executada, porque a dependência de ADD.D em DIV.D faz com que o pipeline fique em stall; mesmo assim, SUB.D não é dependente de dados de qualquer coisa no pipeline. Esse hazard cria uma limitação de desempenho que pode ser eliminada por não exigir que as instruções sejam executadas na ordem do programa.

No pipeline clássico em cinco estágios, os hazards estruturais e de dados poderiam ser verificados durante a decodificação da instrução (ID): quando uma instrução pudesse ser executada sem hazards, ela seria enviada pela ID sabendo que todos os hazards de dados foram resolvidos.

Para que possamos começar a executar o SUB.D no exemplo anterior, temos de dividir o processo em duas partes: verificar quaisquer hazards estruturais e esperar pela ausência de um hazard de dados. Ainda assim usamos o despacho de instruções na ordem (ou seja, instruções enviadas na ordem do programa), mas queremos que uma instrução comece sua execução assim que seus operandos de dados estiverem disponíveis. Esse pipeline realiza a *execução fora de ordem*, que implica em *término fora de ordem*.

A execução fora de ordem introduz a possibilidade de hazards WAR e WAW, que não existem no pipeline de inteiros de cinco estágios, e sua extensão lógica a um pipeline de

ponto flutuante em ordem. Considere a sequência de códigos de ponto flutuante MIPS a seguir:

```
DIV.D     F0,F2,F4
ADD.D     F6,F0,F8
SUB.D     F8,F10,F14
MUL.D     F6,F10,F8
```

Existe uma antidependência entre o ADD.D e o SUB.D e, se o pipeline executar o SUB.D antes do ADD.D (que está esperando por DIV.D), ele violará a antidependência, gerando um hazard WAR. De modo semelhante, para evitar violar as dependências de saída, como a escrita de F6 por MUL.D, os hazards WAW precisam ser tratados. Conforme veremos, esses dois hazards são evitados pelo uso da renomeação de registradores.

O término fora de ordem também cria complicações importantes no tratamento de exceções. O escalonamento dinâmico com o término fora de ordem precisa preservar o comportamento da exceção no sentido de que *exatamente* as exceções que surgiriam se o programa fosse executado na ordem estrita do programa *realmente* surjam. Processadores escalonados dinamicamente preservam o comportamento de exceção garantindo que nenhuma instrução possa gerar uma exceção até que o processador saiba que a instrução que levanta a exceção será executada; veremos brevemente como essa propriedade pode ser garantida.

Embora o comportamento de exceção tenha de ser preservado, os processadores escalonados dinamicamente podem gerar exceções *imprecisas*. Uma exceção é *imprecisa* se o estado do processador quando uma exceção for levantada não se parecer exatamente como se as instruções fossem executadas sequencialmente na ordem estrita do programa. Exceções imprecisas podem ocorrer devido a duas possibilidades:

1. O pipeline pode ter instruções *já completadas*, que estão *mais adiante* na ordem do programa do que a instrução que causa a exceção.
2. O pipeline pode *ainda não ter completado* algumas instruções, que estão *mais atrás* na ordem do programa do que a instrução que causa a exceção.

As exceções imprecisas dificultam o reinício da execução após uma exceção. Em vez de resolver esses problemas nesta seção, discutiremos na Seção 3.6 uma solução que oferece exceções precisas no contexto de um processador com especulação. Para exceções de ponto flutuante, outras soluções foram usadas, conforme discutiremos no Apêndice J.

Para permitir a execução fora de ordem, basicamente dividimos o estágio ID do nosso pipeline simples de cinco estágios em dois estágios:

1. *Despacho.* Decodificar instruções, verificar hazards estruturais.
2. *Leitura de operandos.* Esperar até que não haja hazards de dados, depois ler operandos.

Um estágio de load de instrução precede o estágio de despacho e pode carregar tanto de um registrador de instrução quanto de uma fila de instruções pendentes; as instruções são então enviadas a partir do registrador ou da fila. O estágio EX segue o estágio de leitura de operandos, assim como no pipeline de cinco estágios. A execução pode levar vários ciclos, dependendo da operação.

Distinguimos quando uma instrução *inicia a execução* e quando ela *termina a execução*; entre os dois momentos, a instrução está *em execução*. Nosso pipeline permite que várias instruções estejam em execução ao mesmo tempo e, sem essa capacidade, uma vantagem importante do escalonamento dinâmico é perdida. Ter várias instruções em execução ao mesmo tempo exige várias unidades funcionais, unidades funcionais pipelined ou ambas.

Como essas duas capacidades — unidades funcionais pipelined e múltiplas unidades funcionais — são essencialmente equivalentes para fins de controle de pipeline, vamos considerar que o processador tem várias unidades funcionais.

Em um pipeline escalonado dinamicamente, todas as instruções passam de maneira ordenada pelo estágio de despacho (despacho na ordem); porém, elas podem ser proteladas ou contornadas entre si no segundo estágio (leitura de operandos) e, assim, entrar na execução fora de ordem. O *scoreboarding* é uma técnica para permitir que as instruções sejam executadas fora de ordem quando houver recursos suficientes e nenhuma dependência de dados; recebu esse nome após o CDC 6600 scoreboard, que desenvolveu essa capacidade, e nós a discutiremos no Apêndice A. Aqui, enfocamos uma técnica mais sofisticada, chamada *algoritmo de Tomasulo*, que apresenta várias melhorias importantes em relação ao scoreboarding. Adicionalmente, o algoritmo de Tomasulo pode ser estendido para lidar com *especulação*, uma técnica para reduzir o efeito das dependências de controle prevendo o resultado de um desvio, executando instruções no endereço de destino previsto e realizando ações de previsão quando a previsão estiver incorreta. Embora provavelmente o uso de scoreboarding seja suficiente para suportar um superescalar simples de dois níveis como o ARM A8, um processador mais agressivo, como o Intel i7 de quatro despachos, se beneficia do uso da execução fora de ordem.

Escalonamento dinâmico usando a técnica de Tomasulo

A unidade de ponto flutuante IBM 360/91 usava um esquema sofisticado para permitir a execução fora de ordem. Esse esquema, inventado por Robert Tomasulo, verifica quando os operandos para as instruções estão disponíveis, para minimizar os hazards RAW e introduz a renomeação de registrador para minimizar os hazards WAW e WAR. Nos processadores modernos existem muitas variações desse esquema, embora os principais conceitos do rastreamento de dependência de instrução — para permitir a execução assim que os operandos estiverem disponíveis e a renomeação de registradores para evitar os hazards WAR e WAW — sejam características comuns.

O objetivo da IBM foi conseguir alto desempenho de ponto flutuante a partir de um conjunto de instruções e de compiladores projetados para toda a família de computadores 360, em vez de compiladores especializados para os processadores de ponta. A arquitetura 360 tinha apenas quatro registradores de ponto flutuante de precisão dupla, o que limita a eficácia do escalonamento do compilador; esse fato foi outra motivação para a técnica de Tomasulo. Além disso, o IBM 360/91 tinha longos acessos à memória e longos atrasos de ponto flutuante, o que o algoritmo de Tomasulo foi projetado para contornar. Ao final desta seção, veremos que o algoritmo de Tomasulo também pode admitir a execução sobreposta de várias iterações de um loop.

Explicamos o algoritmo, que enfoca a unidade de ponto flutuante e a unidade de load-store, no contexto do conjunto de instruções do MIPS. A principal diferença entre o MIPS e o 360 é a presença das instruções registrador-memória na segunda arquitetura. Como o algoritmo de Tomasulo utiliza uma unidade funcional de load, nenhuma mudança significativa é necessária para acrescentar os modos de endereçamento registrador-memória. O IBM 360/91 também tinha unidades funcionais pipelined, em vez de múltiplas unidades funcionais, mas descrevemos o algoritmo como se houvesse múltiplas unidades funcionais. Essa é uma extensão conceitual simples para também utilizar o pipeline nessas unidades funcionais.

Conforme veremos, os hazards RAW são evitados executando-se uma instrução apenas quando seus operandos estiverem disponíveis. Hazards WAR e WAW, que surgem das dependências de nomes, são eliminados pela renomeação de registrador. A *renomeação*

de registradores elimina esses hazards renomeando todos os registradores de destino, incluindo aqueles com leitura ou escrita pendente de uma instrução anterior, de modo que a escrita fora de ordem não afeta quaisquer instruções que dependam de um valor anterior de um operando.

Para entender melhor como a renomeação de registradores elimina os hazards WAR e WAW, considere o exemplo de sequência de código a seguir, que inclui um hazard WAR e um WAW em potencial:

```
DIV.D     F0,F2,F4
ADD.D     F6,F0,F8
S.D       F6,0(R1)
SUB.D     F8,F10,F14
MUL.D     F6,F10,F8
```

Existe uma antidependência entre o ADD.D e o SUB.D e uma dependência de saída entre o ADD.D e o MUL.D, levando a dois hazards possíveis, um hazard WAR no uso de F8 por ADD.D e um hazard WAW, pois o ADD.D pode terminar depois do MUL.D. Também existem três dependências de dados verdadeiras: entre o DIV.D e o ADD.D, entre o SUB.D e o MUL.D, e entre o ADD.D e o S.D.

Essas duas dependências de nome podem ser eliminadas pela renomeação de registrador. Para simplificar, considere a existência de dois registradores temporários, S e T. Usando S e T, a sequência pode ser reescrita sem quaisquer dependências como:

```
DIV.D     F0,F2,F4
ADD.D     S,F0,F8
S.D       S,0(R1)
SUB.D     T,F10,F14
MUL.D     F6,F10,T
```

Além disso, quaisquer usos subsequentes de F8 precisam ser substituídos pelo registrador T. Nesse segmento de código, o processo de renomeação pode ser feito estaticamente pelo compilador. A descoberta de quaisquer usos de F8 que estejam mais adiante no código exige análise sofisticada do compilador ou suporte do hardware, pois podem existir desvios entre o segmento de código anterior e um uso posterior de F8. Conforme veremos, o algoritmo de Tomasulo pode lidar com a renomeação entre desvios.

No esquema de Tomasulo, a renomeação de registrador é fornecida por *estações de reserva*, que colocam em buffer os operandos das instruções esperando para serem enviadas. A ideia básica é que uma estação de reserva apanhe e coloque um operando em um buffer assim que ele estiver disponível, eliminando a necessidade de carregar o operando de um registrador. Além disso, instruções pendentes designam a estação de reserva que fornecerá seu suporte. Finalmente, quando escritas sucessivas em um registrador forem superpostas na execução, somente a última será realmente utilizada para atualizar o registrador. À medida que as instruções forem enviadas, os especificadores de registrador para operandos pendentes serão trocados para os nomes da estação de reserva, que oferecerá renomeação de registrador.

Como pode haver mais estações de reserva do que registradores reais, a técnica pode até mesmo eliminar hazards que surgem das dependências de nome que não poderiam ser eliminadas por um compilador. À medida que explorarmos os componentes do esquema de Tomasulo, retornaremos ao tópico de renomeação de registradores e veremos exatamente como ocorre a renomeação e como ela elimina os hazards WAR e WAW.

O uso de estações de reserva, em vez de um banco de registradores centralizado, leva a duas outras propriedades importantes: 1) a detecção de hazard e o controle de execução

são distribuídos: a informação mantida nas estações de reserva em cada unidade funcional determina quando uma instrução pode iniciar a execução nessa unidade; 2) os resultados são passados diretamente para as unidades funcionais a partir das estações de reserva, onde são mantidos em buffer em vez de passarem pelos registradores. Esse bypass é feito com um barramento de resultados comum, que permite que todas as unidades esperando por um operando sejam carregadas simultaneamente (no 360/91, isso é chamado de *barramento de dados comum* ou CDB). Em pipelines com múltiplas unidades de execução e enviando múltiplas instruções por clock, será necessário o uso de mais de um barramento de resultados.

A Figura 3.6 mostra a estrutura básica de um processador baseado em Tomasulo, incluindo a unidade de ponto flutuante e a unidade de load-store; nenhuma das tabelas do controle de execução aparece. Cada estação de reserva mantém uma instrução que foi enviada e está esperando a execução em uma unidade funcional e outros valores operando para essa instrução, se já tiverem sido calculados, ou então os nomes das estações de reserva que oferecerão os valores de operando.

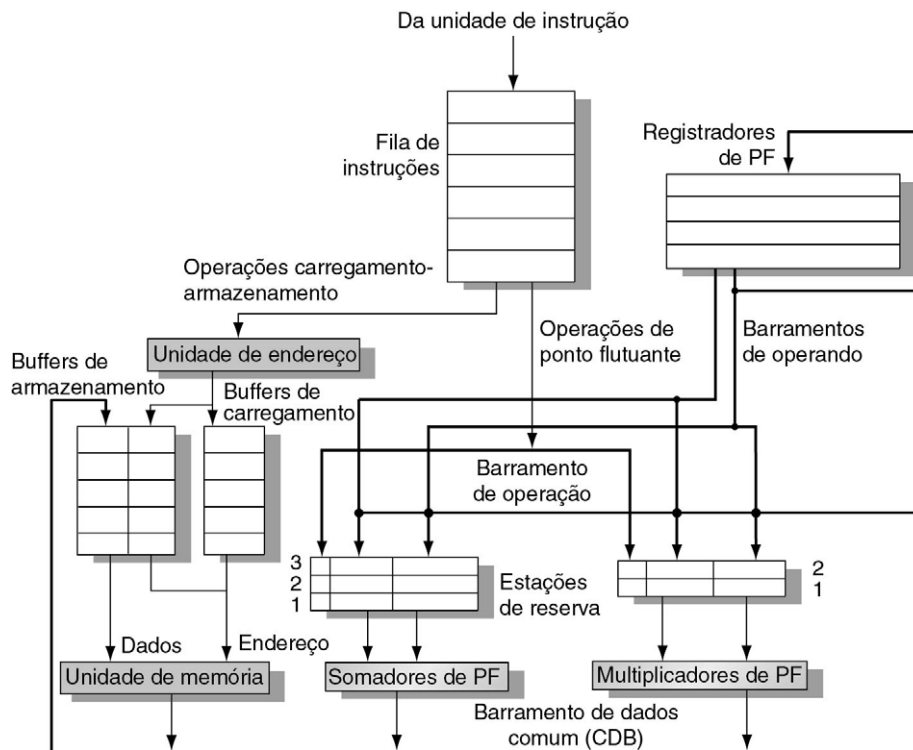



FIGURA 3.6 Estrutura básica de uma unidade de ponto flutuante MIPS usando o algoritmo de Tomasulo. As instruções são enviadas da unidade de instrução para a fila de instruções, da qual são enviadas na ordem FIFO. As estações de reserva incluem a operação e os operandos reais, além das informações usadas para detectar e resolver hazards. Buffers de load possuem três funções: manter os componentes do endereço efetivo até que ele seja calculado, rastrear loads pendentes que estão aguardando na memória e manter os resultados dos loads completados que estão esperando pelo CDB. De modo semelhante, os buffers de store possuem três funções: 1) manter os componentes do endereço efetivo até que ele seja calculado; 2) manter os endereços de memória de destino dos stores pendentes que estão aguardando pelo valor de dado para armazenar; e 3) manter o endereço e o valor a armazenar até que a unidade de memória esteja disponível. Todos os resultados das unidades de PF ou da unidade de load são colocados no CDB, que vai para o banco de registradoreses de PF e também para as estações de reserva e buffers de store. Os somadores de PF implementam adição e subtração, e os multiplicadores de PF realizam a multiplicação e a divisão.

Os buffers de load e os buffers de store mantêm dados ou endereços vindo e indo para a memória, comportando-se quase exatamente como estações de reserva, de modo que os distinguimos somente quando necessário. Os registradores de ponto flutuante estão conectados por um par de barramentos para as unidades funcionais e por um único barramento para os buffers de store. Todos os resultados das unidades funcionais e da memória são enviados no barramento de dados comum, que vai para toda parte, exceto para o buffer de load. Todas as estações de reserva possuem campos de tag, empregados pelo controle do pipeline.

Antes de descrevermos os detalhes das estações de reserva e do algoritmo, vejamos as etapas pelas quais uma instrução passa. Existem apenas três etapas, embora cada uma possa usar um número arbitrário de ciclos de clock:

1. *Despacho*. Carregue a próxima instrução do início da fila de instrução, que é mantida na ordem FIFO para garantir a manutenção do fluxo de dados correto. Se houver determinada estação de reserva que esteja vazia, a instrução será enviada para a estação com os valores de operando, se estiverem nos registradores. Se não houver uma estação de reserva vazia, haverá um hazard estrutural, e a instrução ficará em stall até que uma estação ou um buffer seja liberado. Se os operandos não estiverem nos registradores, registre as unidades funcionais que produzirão os operandos. Essa etapa renomeia registradores, eliminando os hazards WAR e WAW. (Esse estágio, às vezes, é chamado de *despacho* em um processador com escalonamento dinâmico.)
2. *Execução*. Se um ou mais operandos ainda não estiver disponível, monitore o barramento de dados comum enquanto espera que ele seja calculado. Quando um operando estiver disponível, ele será colocado em qualquer estação de reserva que o esperar. Quando todos os operandos estiverem disponíveis, a operação poderá ser executada na unidade funcional correspondente. Adiando a execução da instrução até que os operandos estejam disponíveis, os hazards RAW serão evitados. (Alguns processadores com escalonamento dinâmico chamam essa etapa de "despacho", mas usamos o termo "execução", que foi usado no primeiro processador com escalonamento dinâmico, o CDC 6600.)

   Observe que várias instruções poderiam ficar prontas no mesmo ciclo de clock para a mesma unidade funcional. Embora as unidades funcionais independentes possam iniciar a execução no mesmo ciclo de clock para diferentes instruções, se mais de uma instrução estiver pronta para uma única unidade funcional a unidade terá de escolher entre elas. Para as estações de reserva de ponto flutuante, essa escolha pode ser feita arbitrariamente; porém, loads e stores apresentam uma complicação adicional.

   Loads e stores exigem um processo de execução em duas etapas. A primeira etapa calcula o endereço efetivo quando o registrador de base estiver disponível, e então o endereço efetivo é colocado no buffer de load ou store. Loads no buffer de load são executados assim que a unidade de memória está disponível. Stores no buffer de store esperam pelo valor a ser armazenado antes de serem enviados à unidade de memória. Loads e stores são mantidos na ordem do programa por meio do cálculo do endereço efetivo, que ajudará a impedir problemas na memória, conforme veremos em breve.

   Para preservar o comportamento da exceção, nenhuma instrução tem permissão para iniciar sua execução até que todos os desvios que precedem a instrução na ordem do programa tenham sido concluídos. Essa restrição garante que uma instrução que causa uma exceção durante a execução realmente tenha sido executada. Em um processador usando a previsão de desvio (como é feito em todos os processadores com escalonamento dinâmico), isso significa que o processador precisa saber que a previsão de desvio

estava correta antes de permitir o início da execução de uma instrução após o desvio. Se o processador registrar a ocorrência da exceção, mas não a tratar de fato, uma instrução poderá iniciar sua execução mas não ser protelada até que entre na escrita do resultado.

Conforme veremos, a especulação oferece um método mais flexível e mais completo para lidar com as exceções. Por isso, deixaremos essa melhoria para depois, a fim de mostrarmos como a especulação trata desse problema.

3. *Escrita do resultado.* Quando o resultado estiver disponível, escreva-o no CDB e, a partir daí, nos registradores e em quaisquer estações de reserva (incluindo os buffers de store) esperando por esse resultado. Os stores são mantidos no buffer de store até que o valor a ser armazenado e o endereço do store estejam disponíveis, e depois o resultado é escrito assim que a unidade de memória ficar livre.

As estruturas de dados que detectam e eliminam os hazards estão conectadas às estações de reserva, ao banco de registradoreses e aos buffers de load e store com informações ligeiramente diferentes conectadas a diferentes objetos. Essas tags são essencialmente nomes para um conjunto estendido de registradores virtuais usados para renomeação. Em nosso exemplo, o campo de tag é uma quantidade de 4 bits que indica uma das cinco estações de reserva ou um dos cinco buffers de load. Como veremos, isso produz o equivalente a 10 registradores que podem ser designados como registradores de resultado (ao contrário dos quatro registradores de precisão dupla que a arquitetura 360 contém). Em um processador com mais registradores reais, desejaríamos que a renomeação fornecesse um conjunto ainda maior de registradores virtuais. O campo de tag descreve qual estação de reserva contém a instrução que produzirá um resultado necessário como operandos-fonte.

Quando uma instrução tiver sido enviada e estiver aguardando um operando-fonte, ela se referirá ao operando pelo número da estação de reserva, atribuída à instrução que escreverá no registrador. Valores não usados, como zero, indicam que o operando já está disponível nos registradores. Como existem mais estações de reserva do que números de registrador reais, os hazards WAW e WAR são eliminados pela renomeação de resultados usando números de estações de reserva. Embora no esquema de Tomasulo as estações de reserva sejam usadas como registradores virtuais estendidos, outras técnicas poderiam usar um conjunto de registradores com registradores adicionais ou uma estrutura como o buffer de reordenação, que veremos na Seção 3.6.

No esquema de Tomasulo, além dos métodos subsequentes que veremos para dar suporte à especulação, os resultados são transmitidos por broadcast a um barramento (o CDB), que é monitorado pelas estações de reserva. A combinação do barramento de resultados comum e da recuperação dos resultados do barramento pelas estações de reserva implementa os mecanismos de encaminhamento e bypass usados em um pipeline escalonado estaticamente. Porém, ao fazer isso, um esquema escalonado dinamicamente introduz um ciclo de latência entre a fonte e o resultado, pois a combinação de um resultado e seu uso não pode ser feita antes do estágio de escrita do resultado. Assim, em um pipeline escalonado dinamicamente, a latência efetiva entre uma instrução produzindo e uma instrução consumindo é pelo menos um ciclo maior que a latência da unidade funcional que produz o resultado.

É importante lembrar que as tags no esquema de Tomasulo se referem ao buffer ou unidade que vai produzir um resultado. Os nomes de registrados são descartados quando uma instrução envia para uma estação de reserva. (Essa é uma das diferenças-chave entre o esquema de Tomasulo e o scoreboarding: no scoreboarding, os operandos permanecem nos registradores e somente são lidos depois da instrução que os produziu ser completada e da instrução que vai consumi-lo estar pronta para ser executada.)

Cada estação de reserva possui sete campos:

- Op. A operação a ser realizada sobre os operandos-fonte S1 e S2.
- Qj, Qk. As estações de reserva que produzirão o operandos-fonte correspondentes; um valor zero indica que o operando-fonte já está disponível em Vj ou Vk, ou é desnecessário. (O IBM 360/91 os chama SINKunit e SOURCEunit.)
- Vj, Vk. O valor dos operandos-fonte. Observe que somente o campo V ou o campo Q é válido para cada operando. Para loads, o campo Vk é usado para manter o campo de offset. (Esses campos são chamados SINK e SOURCE no IBM 360/91.)
- A. Usado para manter informações para o cálculo de endereço de memória para um load ou store. Inicialmente, o campo imediato da instrução é armazenado aqui; após o cálculo do endereço, o endereço efetivo é armazenado aqui.
- Busy. Indica que essa estação de reserva e sua resepctiva unidade funcional estão ocupadas.

O banco de registradores possui um campo, Qi:

- Qi. O número da estação de reserva que contém a operação cujo resultado deve ser armazenado nesse registrador. Se o valor de Qi estiver em branco (ou 0), nenhuma instrução atualmente ativa está calculando um resultado destinado a esse registrador, significando que o valor é simplesmente o conteúdo do registrador.

Os buffers de load e store possuem um campo cada, A, que mantém o resultado do endereço efetivo quando a primeira etapa da execução tiver sido concluída.

Na próxima seção, primeiro vamos considerar alguns exemplos que mostram como funcionam esses mecanismos e depois examinaremos o algoritmo detalhado.

3.5 ESCALONAMENTO DINÂMICO: EXEMPLOS E ALGORITMO

Antes de examinarmos o algoritmo de Tomasulo com detalhes, vamos considerar alguns exemplos que ajudarão a ilustrar o modo como o algoritmo funciona.

Exemplo Mostre como se parecem as tabelas de informação para a sequência de código a seguir quando somente o primeiro load tiver sido concluído e seu resultado escrito:

```
1.    L.D      F6,32(R2)
2.    L.D      F2,44(R3)
3.    MUL.D    F0,F2,F4
4.    SUB.D    F8,F2,F6
5.    DIV.D    F10,F0,F6
6.    ADD.D    F6,F8,F2
```

Resposta A Figura 3.7 mostra o resultado em três tabelas. Os números anexados aos nomes add, mult e load indicam a tag para a estação de reserva — Add1 é a tag para o resultado da primeira unidade de soma. Além disso, incluímos uma tabela de status de instrução. Essa tabela foi incluída apenas para ajudá-lo a entender o algoritmo; ela *não* faz parte do hardware. Em vez disso, a estação de reserva mantém o status de cada operação que foi enviada.

O esquema de Tomasulo oferece duas vantagens importantes e mais simples em relação aos esquemas anteriores: 1) a distribuição da lógica de detecção de hazard e 2) a eliminação de stalls para hazards WAW e WAR.

A primeira vantagem surge das estações de reserva distribuídas e do uso do Common Data Bus (CDB). Se várias instruções estiverem aguardando um único resultado e cada instrução

Instrução	Status da instrução		
	Despacho	Execução	Escrita de resultado
L.D F6,32(R2)	√	√	√
L.D F2,44(R3)	√	√	
MUL.D F0,F2,F4	√		
SUB.D F8,F2,F6	√		
DIV.D F10,F0,F6	√		
ADD.D F6,F8,F2	√		

	Estações de reserva						
Nome	Busy	Op	Vj	Vk	Qj	Qk	A
Load1	não						
Load2	sim	Load					44 + Regs[R3]
Add1	sim	SUB		Mem[32 + Regs[R2]]	Load2		
Add2	sim	ADD			Add1	Load2	
Add3	não						
Mult1	sim	MUL		Regs[F4]	Load2		
Mult2	sim	DIV		Mem[32 + Regs[R2]]	Mult1		

	Status dos registradores							
Campo	F0	F2	F4	F6	F8	F10	F12	...	F30
Qi	Mult1	Load2		Add2	Add1	Mult2			

FIGURA 3.7 Estações de reserva e tags de registradores mostradas quando todas as instruções forem enviadas, mas somente a primeira instrução load tiver sido concluída e seu resultado escrito no CDB.

O segundo load concluiu o cálculo do endereço efetivo, mas está esperando na unidade de memória. Usamos o array Registros[ ] para nos referirmos ao banco de registradores, e o array Mem[ ] para nos referirmos à memória. Lembre-se de que um operando é especificado por um campo Q ou um campo V a qualquer momento. Observe que a instrução ADD.D, que tem um hazard WAR no estágio WB, foi enviada e poderia ser concluída antes que o DIV.D se inicie.

já tiver seu outro operando, as instruções poderão ser liberadas simultaneamente por broadcast do resultado no CDB. Se um banco de registradores centralizado fosse utilizado, as unidades teriam de ler seus resultados dos registradores quando os barramentos de registrador estivessem disponíveis.

A segunda vantagem, a eliminação de hazards WAW e WAR, é obtida renomeando-se os registradores por meio das estações de reserva e pelo processo de armazenar operandos na estação de reserva assim que estiverem disponíveis.

Por exemplo, a sequência de código na Figura 3.7 envia o DIV.D e o ADD.D, embora exista um hazard WAR envolvendo F6. O hazard pode ser eliminado de duas maneiras. Primeiro, se a instrução oferecendo o valor para o DIV.D tiver sido concluída, Vk armazenará o resultado, permitindo que DIV.D seja executado independentemente do ADD.D (esse é o caso mostrado). Por outro lado, se o L.D não tivesse sido concluído, Qk apontaria para a estação de reserva Load1 e a instrução DIV.D seria independente do ADD.D. Assim, de qualquer forma, o ADD.D pode ser enviado e sua execução iniciada. Quaisquer usos do resultado do DIV.D apontariam para a estação de reserva,

permitindo que o ADD.D concluísse e armazenasse seu valor nos registradores sem afetar o DIV.D.

Veremos um exemplo da eliminação de um hazard WAW em breve. Mas primeiro vejamos como nosso exemplo anterior continua a execução. Nesse exemplo, e nos exemplos seguintes dados neste capítulo, consideramos estas latências: load usa um ciclo de clock, uma adição usa dois ciclos de clock, multiplicação usa seis ciclos de clock e divisão usa 12 ciclos de clock.

Exemplo Usando o mesmo segmento de código do exemplo anterior (página 152), mostre como ficam as tabelas de status quando o MUL.D está pronto para escrever seu resultado.

Resposta O resultado aparece nas três tabelas da Figura 3.8. Observe que ADD.D foi concluída, porque os operandos de DIV.D foram copiados, contornando assim o hazard WAR. Observe que, mesmo que o load de F6 fosse adiado, o add em F6 poderia ser executado sem disparar um hazard WAW.

Algoritmo de Tomasulo: detalhes

A Figura 3.9 especifica as verificações e etapas pelas quais cada instrução precisa passar. Como já dissemos, loads e stores passam por uma unidade funcional para cálculo de endereço efetivo antes de prosseguirem para buffers de load e store independentes. Os loads usam uma segunda etapa de execução para acessar a memória e depois passar para

	Status da instrução		
Instrução	**Despacho**	**Execução**	**Escrita de resultado**
`L.D F6,32(R2)`	√	√	√
`L.D F2,44(R3)`	√	√	√
`MUL.D F0,F2,F4`	√	√	
`SUB.D F8,F2,F6`	√	√	√
`DIV.D F10,F0,F6`	√		
`ADD.D F6,F8,F2`	√	√	√

	Estações de reserva						
Nome	**Busy**	**Op**	**Vj**	**Vk**	**Qj**	**Qk**	**A**
Load1	Não						
Load2	Não						
Add1	Não						
Add2	Não						
Add3	Não						
Mult1	Sim	MUL	`Mem[44 + Regs[R3]]`	`Regs[F4]`			
Mult2	Sim	DIV		`Mem[32 + Regs[R2]]`	Mult1		

	Status dos registradores							
Campo	**F0**	**F2**	**F4**	**F6**	**F8**	**F10**	**F12**	**. . .**	**F30**
Qi	Mult1					Mult2			

FIGURA 3.8 Multiplicação e divisão são as únicas instruções não terminadas.

Status da instrução	Esperar até	Ação ou manutenção

Despacho
Operação de PF
	Estação *r* vazia	if (RegisterStat[rs].Qi¦0)
		{RS[r].Qj ← RegisterStat[rs].Qi}
		else {RS[r].Vj ← Regs[rs]; RS[r].Qj ← 0};
		if (RegisterStat[rt].Qi¦0)
		{RS[r].Qk ← RegisterStat[rt].Qi
		else {RS[r].Vk ← Regs[rt]; RS[r].Qk ← 0};
		RS[r].Busy ← yes; RegisterStat[rd].Q ← r;

Load ou store	Buffer *r* vazio	if (RegisterStat[rs].Qi¦0) {RS[r].Qj ← RegisterStat[rs].Qi} else {RS[r].Vj ← Regs[rs]; RS[r].Qj ← 0}; RS[r].A ← imm; RS[r].Busy ← yes;
Apenas load		RegisterStat[rt].Qi ← r;
Apenas store		if (RegisterStat[rt].Qi¦0) {RS[r].Qk ← RegisterStat[rs].Qi} else {RS[r].Vk ← Regs[rt]; RS[r].Qk ← 0};
Execução Operação de PF	(RS[r].Qj = 0) e (RS[r].Qk = 0)	Calcular resultados: operandos estão em Vj e Vk
Load-store etapa 1	RS[r].Qj = 0 & r é o início da fila de store	RS[r].A ← RS[r].Vj + RS[r].A;
Load etapa 2	Load da etapa 1 concluído	Ler de Mem[RS[r].A]
Escrever resultado Operação de PF ou load	Execução completa em r & CDB disponível	∀x(if (RegisterStat[x].Qi=r) {Regs[x] ← result; RegisterStat[x].Qi ← 0}); ∀x(if (RS[x].Qj=r) {RS[x].Vj ← result;RS[x].Qj ← 0}); ∀x(if (RS[x].Qk=r) {RS[x].Vk ← result;RS[x].Qk ← 0}); RS[r].Busy ← no;
Store	Execução completa em r & RS[r].Qk = 0	Mem[RS[r].A] ← RS[r].Vk; RS[r].Busy ← no;

FIGURA 3.9 **Etapas no algoritmo e o que é exigido para cada etapa.**

Para a instrução sendo enviada, rd é o destino, rs e rt são os números dos registrador fontes, imm é o campo imediato com extensão de sinal e *r* é a estação de reserva ou buffer ao qual a instrução está atribuída. RS é a estrutura de dados da estação de reserva. O valor retornado por uma unidade de PF ou pela unidade de load é chamado de result. RegisterStat é a estrutura de dados de status do registrador (não o banco de registradores, que é Regs[ ]). Quando uma instrução é enviada, o registrador de destino tem seu campo Qi definido com o número do buffer ou da estação de reserva à qual a instrução é enviada. Se os operandos estiverem disponíveis nos registradores, eles serão armazenados nos campos V. Caso contrário, os campos Q serão definidos para indicar a estação de reserva que produzirá os valores necessários como operandos-fontes. A instrução espera na estação de reserva até que seus dois operandos estejam disponíveis, indicado por zero nos campos Q. Os campos Q são definidos com zero quando essa instrução é enviada ou quando uma instrução da qual essa instrução depende é concluída e realiza sua escrita de volta. Quando uma instrução tiver terminado sua execução e o CDB estiver disponível, ela poderá realizar sua escrita de volta. Todos os buffers, registradores e estações de reserva cujo valor de Qj ou Qk é igual à estação de reserva concluída atualizam seus valores pelo CDB e marcam os campos Q para indicar que os valores foram recebidos. Assim, o CDB pode transmitir seu resultado por broadcast para muitos destinos em um único ciclo de clock e, se as instruções em espera tiverem seus operandos, elas podem iniciar sua execução no próximo ciclo de clock. Loads passam por duas etapas na Execução, e os stores funcionam um pouco diferente durante a escrita de resultados, podendo ter de esperar pelo valor a armazenar. Lembre-se de que, para preservar o comportamento da exceção, as instruções não têm permissão para serem executadas se um desvio anterior na ordem do programa não tiver sido concluído. Como qualquer conceito de ordem de programa não é mantido após o estágio de despacho, essa restrição normalmente é implementada impedindo-se que qualquer instrução saia da etapa de despacho, se houver um desvio pendente já no pipeline. Na Seção 3.6, veremos como o suporte à especulação remove essa restrição.

o estágio de escrita de resultados, a fim de enviar o valor da memória para o banco de registradores e/ou quaisquer estações de reserva aguardando. Os stores completam sua execução no estágio de escrita de resultados, que escreve o resultado na memória. Observe que todas as escritas ocorrem nesse estágio, seja o destino um registrador ou a memória. Essa restrição simplifica o algoritmo de Tomasulo e é fundamental para a sua extensão com especulação na Seção 3.6.

Algoritmo de Tomasulo: exemplo baseado em loop

Para entender o poder completo da eliminação de hazards WAW e WAR por meio da renomeação dinâmica de registradores, temos de examinar um loop. Considere a sequência simples ab seguir para multiplicar os elementos de um array por um escalar em F2:

```
Loop:   L.D      F0,0(R1)
        MUL.D    F4,F0,F2
        S.D      F4,0(R1)
        DADDIU   R1,R1,-8
        BNE      R1,R2,Loop; desvia se R1€R2
```

Se prevermos que os desvios serão tomados, o uso de estações de reserva permitirá que várias execuções desse loop prossigam ao mesmo tempo. Essa vantagem é obtida sem mudar o código — de fato, o loop é desdobrado dinamicamente pelo hardware, usando as estações de reserva obtidas pela renomeação para atuar como registradores adicionais.

Vamos supor que tenhamos enviado todas as instruções em duas iterações sucessivas do loop, mas nenhum dos loads-stores ou operações de ponto flutuante tenham sido concluídas. A Figura 3.10 mostra as estações de reserva, tabelas de status de registrador e buffers de load e store nesse ponto (a operação da ALU com inteiros é ignorada e considera-se

	Status da instrução			
Instrução	**Da iteração**	**Despacho**	**Execução**	**Escrita de resultado**
L.D F0,0(R1)	1	√	√	
L.D F4,F0,F2	1	√		
MUL.D F4,0(R1)	1	√	√	
SUB.D F0,0(R1)	2	√		
DIV.D F4,F0,F2	2	√		
ADD.D F4,0(R1)	2	√		

	Estações de reserva						
Nome	**Busy**	**Op**	**Vj**	**Vk**	**Qj**	**Qk**	**A**
Load1	sim	Load					Regs[R1] + 0
Load2	sim	Load					Regs[R1] - 8
Add1	não						
Add2	não						
Add3	não						
Mult1	sim	MUL		Regs[F2]	Load1		
Mult2	sim	MUL		Regs[F2]	Load2		
Store1	sim	Store	Regs[R1]			Mult1	
Store2	sim	Store	Regs[R1] - 8			Mult2	

	Status dos registradores								
Campo	**F0**	**F2**	**F4**	**F6**	**F8**	**F10**	**F12**	**. . .**	**F30**
Qi	Load2		Mult2						

FIGURA 3.10 Duas iterações ativas do loop sem qualquer instrução concluída.
As entradas nas estações de reserva do multiplicador indicam que os loads pendentes são as fontes. As estações de reserva do store indicam que o destino da multiplicação é a fonte do valor a armazenar.

que o desvio foi previsto como sendo tomado). Quando o sistema alcança esse estado, duas cópias do loop poderiam ser sustentadas com um CPI perto de 1,0, desde que as multiplicações pudessem ser completadas em quatro ciclos de clock. Com uma latência de seis ciclos, iterações adicionais terão de ser processadas antes que o estado seguro possa ser alcançado. Isso exige mais estações de reserva para manter as instruções que estão em execução.

Conforme veremos mais adiante neste capítulo, quando estendida com o múltiplo despacho de instruções, a técnica de Tomasulo pode sustentar mais de uma instrução por clock.

Um load e um store podem seguramente ser feitos fora de ordem, desde que acessem diferentes endereços. Se um load e um store acessarem o mesmo endereço, então:

- o load vem antes do store na ordem do programa e sua inversão resulta em um hazard WAR ou
- o store vem antes do load na ordem do programa e sua inversão resulta em um hazard RAW.

De modo semelhante, a inversão de dois stores para o mesmo endereço resulta em um hazard WAW.

Logo, para determinar se um load pode ser executado em certo momento, o processador pode verificar se qualquer store não concluído que precede o load na ordem do programa compartilha o mesmo endereço de memória de dados que o load. De modo semelhante, um store precisa esperar até que não haja loads ou stores não executados que estejam antes, na ordem do programa, e que compartilham o mesmo endereço de memória de dados. Consideramos um método para eliminar essa restrição na Seção 3.9.

Para detectar tais hazards, o processador precisa ter calculado o endereço de memória de dados associado a qualquer operação de memória anterior. Uma maneira simples, porém não necessariamente ideal, de garantir que o processador tenha todos esses endereços é realizar os cálculos de endereço efetivo na ordem do programa. (Na realidade, só precisamos manter a ordem relativa entre os stores e outras referências de memória, ou seja, os loads podem ser reordenados livremente.)

Vamos considerar a situação de um load primeiro. Se realizarmos o cálculo de endereço efetivo na ordem do programa, quando um load tiver completado o cálculo de endereço efetivo poderemos verificar se existe um conflito de endereço examinando o campo A de todos os buffers de store ativos. Se o endereço de load combinar com o endereço de quaisquer entradas ativas no buffer de store, essa instrução load não será enviada ao buffer de load até que o store em conflito seja concluído. (Algumas implementações contornam o valor diretamente para o load a partir de um store pendente, reduzindo o atraso para esse hazard RAW.)

Os stores operam de modo semelhante, exceto pelo fato de que o processador precisa verificar os conflitos nos buffers de load e nos buffers de store, pois os stores em conflito não podem ser reordenados com relação a um load ou a um store.

Um pipeline com escalonamento dinâmico pode gerar um desempenho muito alto, desde que os desvios sejam previstos com precisão — uma questão que resolveremos na última seção. A principal desvantagem dessa técnica é a complexidade do esquema de Tomasulo, que exige grande quantidade de hardware. Em particular, cada estação de reserva deve conter um buffer associativo, que precisa ser executado em alta velocidade, além da lógica de controle complexa. O desempenho também pode ser limitado pelo

único CDB. Embora CDBs adicionais possam ser incluídos, cada CDB precisa interagir com cada estação de reserva, e o hardware de verificação de tag associativo precisará ser duplicado em cada estação para cada CDB.

No esquema de Tomasulo, duas técnicas diferentes são combinadas: a renomeação dos registradores de arquitetura para um conjunto maior de registradores e a manutenção em buffer dos operandos-fonte a partir do banco de registradores. A manutenção em buffer de operandos-fonte resolve os hazards WAR que surgem quando o operando está disponível nos registradores. Como veremos mais adiante, também é possível eliminar os hazards WAR renomeando um registrador junto com a manutenção de um resultado em buffer até que não haja mais qualquer referência pendente à versão anterior do registrador. Essa técnica será usada quando discutirmos sobre a especulação de hardware.

O esquema de Tomasulo ficou sem uso por muitos anos após o 360/91, mas nos anos 1990 foi bastante adotado nos processadores de múltiplo despacho, por vários motivos:

1. Embora o algoritmo de Tomasulo fosse projetado antes das caches, a presença de caches, com os atrasos inerentemente imprevisíveis, tornou-se uma das principais motivações para o escalonamento dinâmico. A execução fora de ordem permite que os processadores continuem executando instruções enquanto esperam o término de uma falta de cache, escondendo o, assim, toda a penalidade da falta de cache ou parte dela.
2. À medida que os processadores se tornam mais agressivos em sua capacidade de despacho e os projetistas se preocupam com o desempenho de código difícil de escalonamento (como a maioria dos códigos não numéricos), as técnicas como renomeação de registradores e escalonamento dinâmico se tornam mais importantes.
3. Ele pode alcançar alto desempenho sem exigir que o compilador destine o código a uma estrutura de pipeline específica, uma propriedade valiosa na era do software “enlatado” para o mercado em massa.

3.6 ESPECULAÇÃO BASEADA EM HARDWARE

À medida que tentamos explorar mais paralelismo em nível de instrução, a manutenção de dependências de controle se torna um peso cada vez maior. A previsão de desvio reduz os stalls diretos atribuíveis aos desvios, mas, para um processador executando múltiplas instruções por clock, apenas prever os desvios com exatidão pode não ser suficiente para gerar a quantidade desejada de paralelismo em nível de instrução. Um processador de alta capacidade de despacho pode ter de executar um desvio a cada ciclo de clock para manter o desempenho máximo. Logo, a exploração de mais paralelismo requer que contornemos a limitação da dependência de controle.

Contornar a dependência de controle é algo feito especulando o resultado dos desvios e executando o programa como se nossas escolhas fossem corretas. Esse mecanismo representa uma extensão sutil, porém importante, em relação à previsão de desvio com escalonamento dinâmico. Em particular com a especulação, buscamos, enviamos e *executamos* instruções, como se nossas previsões de desvio sempre estivessem corretas; o escalonamento dinâmico só busca e envia essas instruções. Naturalmente, precisamos de mecanismos para lidar com a situação em que a especulação está incorreta. O Apêndice H discute uma série de mecanismos para dar suporte à especulação pelo compilador. Nesta seção, exploraremos a *especulação do hardware*, que estende as ideias do escalonamento dinâmico.

A especulação baseada no hardware combina três ideias fundamentais: 1) previsão dinâmica de desvio para escolher quais instruções executar; 2) especulação para permitir a execução de instruções antes que as dependências de controle sejam resolvidas (com a capacidade de desfazer os efeitos de uma sequência especulada incorretamente); e 3) escalonamento dinâmico para lidar com o escalonamento de diferentes combinações de blocos básicos. (Em comparação, o escalonamento dinâmico sem especulação só sobrepõe parcialmente os blocos básicos, pois exige que um desvio seja resolvido antes de realmente executar quaisquer instruções no bloco básico seguinte.)

A especulação baseada no hardware segue o fluxo previsto de valores de dados para escolher quando executar as instruções. Esse método de executar programas é essencialmente uma *execução de fluxo de dados*: as operações são executadas assim que seus operandos ficam disponíveis.

Para estender o algoritmo de Tomasulo para dar suporte à especulação, temos de separar o bypass dos resultados entre as instruções, que é necessário para executar uma instrução especulativamente, desde o término real de uma instrução. Fazendo essa separação, podemos permitir que uma instrução seja executada e enviar seus resultados para outras instruções, sem possibilitar a ela realizar quaisquer atualizações que não possam ser desfeitas, até sabermos que essa instrução não é mais especulativa.

Usar o valor bypassed é como realizar uma leitura especulativa de registrador, pois não sabemos se a instrução que fornece o valor do registrador-fonte está fornecendo o resultado correto até que a instrução não seja mais especulativa. Quando uma instrução não é mais especulativa, permitimos que ela atualize o banco de registradores ou a memória; chamamos essa etapa adicional na sequência de execução da instrução de *confirmação de instrução* (*instruction commit*).

A ideia central por trás da implementação da especulação é permitir que as instruções sejam executadas fora de ordem, mas forçando-as a serem confirmadas *em ordem* e impedir qualquer ação irrevogável (como atualizar o status ou apanhar uma exceção) até que uma instrução seja confirmada. Logo, quando acrescentamos a especulação, precisamos separar os processos de concluir a execução e confirmar a instrução, pois as instruções podem terminar a execução consideravelmente antes de estarem prontas para confirmar. A inclusão dessa fase de confirmação na sequência de execução da instrução requer um conjunto adicional de buffers de hardware que mantenham os resultados das instruções que terminaram a execução mas não foram confirmadas. Esse buffer de hardware, que chamamos *buffer de reordenação*, também é usado para passar resultados entre instruções que podem ser especuladas.

O buffer de reordenação (ROB) oferece registradores adicionais da mesma forma que as estações de reserva no algoritmo de Tomasulo estendem o conjunto de registradores. O ROB mantém o resultado de uma instrução entre o momento em que a operação associada à instrução termina e o momento em que a instrução é confirmada. Logo, o ROB é a fonte dos operandos para as instruções, assim como as estações de reserva oferecem operandos no algoritmo de Tomasulo. A principal diferença é que, no algoritmo de Tomasulo, quando uma instrução escreve seu resultado, quaisquer instruções enviadas depois disso encontram o resultado no banco de registradores. Com a especulação, o banco de registradores não é atualizado até que a instrução seja confirmada (e nós sabemos que a instrução deverá ser executada); assim, o ROB fornece operandos no intervalo entre o término da execução da instrução e a confirmação da instrução. O ROB é semelhante ao buffer de store no algoritmo de Tomasulo, e integramos a função do buffer de store no ROB para simplificar.

Cada entrada no ROB contém quatro campos: o tipo de instrução, o campo de destino, o campo de valor e o campo de pronto (ready). O campo de tipo de instrução indica

se a instrução é um desvio (e não possui resultado de destino), um store (que tem um endereço de memória como destino) ou uma operação de registrador (operação da ALU ou load, que possui como destinos registradores). O campo de destino fornece o número do registrador (para loads e operações da ALU) ou o endereço de memória (para stores) onde o resultado da instrução deve ser escrito. O campo de valor é usado para manter o valor do resultado da instrução até que a instrução seja confirmada. Veremos um exemplo de entradas ROB em breve. Finalmente, o campo de pronto indica que a instrução completou sua execução, e o valor está pronto.

A Figura 3.11 mostra a estrutura de hardware do processador incluindo o ROB. O ROB substitui os buffers de store. Os stores ainda são executados em duas etapas, mas a segunda etapa é realizada pela confirmação da instrução. Embora a função restante das estações de reserva seja substituída pelo ROB, ainda precisamos de um lugar (buffer) para colocar operações (e operandos) entre o momento em que são enviadas e o momento em que iniciam sua execução. Essa função ainda é fornecida pelas estações de reserva. Como cada instrução tem uma posição no ROB até que seja confirmada, identificamos um resultado

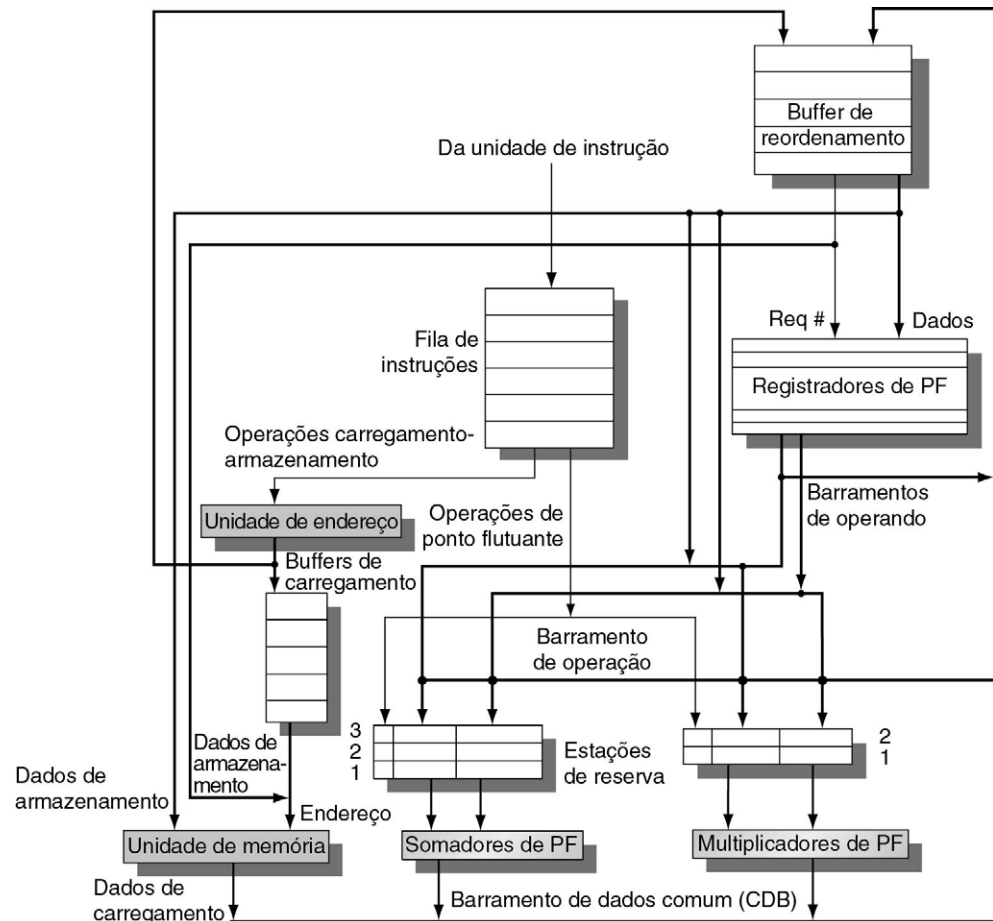



FIGURA 3.11 Estrutura básica de uma unidade de PF usando o algoritmo de Tomasulo e estendida para lidar com a especulação.
Comparando esta figura com a Figura 3.6, na página 149, que implementava o algoritmo de Tomasulo, as principais mudanças são o acréscimo do ROB e a eliminação do buffer de store, cuja função está integrada ao ROB. Esse mecanismo pode ser estendido para o múltiplo despacho, tornando o CDB mais largo para permitir múltiplos términos por clock.

usando o número de entrada do ROB em vez do número da estação de reserva. Essa marcação exige que o ROB atribuído para uma instrução seja rastreado na estação de reserva. Mais adiante nesta seção, exploraremos uma implementação alternativa que usa registradores extras para renomeação e o ROB apenas para rastrear quando as instruções podem ser confirmadas.

Aqui estão as quatro etapas envolvidas na execução da instrução:

1. *Despacho.* Apanhe uma instrução da fila de instruções. Envie a instrução se houver uma estação de reserva vazia e um slot vazio no ROB; envie os operandos à estação de reserva se eles estiverem disponíveis nos registradores ou no ROB. Atualize as entradas de controle para indicar que os buffers estão em uso. O número da entrada do ROB alocada para o resultado também é enviado à estação de reserva, de modo que o número possa ser usado para marcar o resultado quando ele for colocado no CDB. Se todas as reservas estiverem cheias ou o ROB estiver cheio, o despacho de instrução é adiado até que ambos tenham entradas disponíveis.
2. *Execução.* Se um ou mais dos operandos ainda não estiver disponível, monitore o CDB enquanto espera que o registrador seja calculado. Essa etapa verifica os hazards RAW. Quando os dois operandos estiverem disponíveis em uma estação de reserva, execute a operação. As instruções podem levar vários ciclos de clock nesse estágio, e os loads ainda exigem duas etapas nesse estágio. Os stores só precisam ter o registrador de base disponível nessa etapa, pois a execução para um store nesse ponto é apenas o cálculo do endereço efetivo.
3. *Escrita de resultado.* Quando o resultado estiver disponível, escreva-o no CDB (com a tag ROB enviada quando a instrução for enviada) e do CDB para o ROB, e também para quaisquer estações de reserva esperando por esse resultado. Marque a estação de reserva como disponível. Ações especiais são necessárias para armazenar instruções. Se o valor a ser armazenado estiver disponível, ele é escrito no campo Valor da entrada do ROB para o store. Se o valor a ser armazenado ainda não estiver disponível, o CDB precisa ser monitorado até que esse valor seja transmitido, quando o campo Valor na entrada do ROB para o store é atualizado. Para simplificar, consideramos que isso ocorre durante o estágio de escrita de resultado de um store; mais adiante, discutiremos o relaxamento desse requisito.
4. *Confirmação* (*commit*). Esse é o estágio final para o término de uma instrução, após o qual somente seu resultado permanece (alguns processadores chamam essa fase de "término" ou "graduação"). Existem três sequências de ações diferentes na confirmação, dependendo da instrução confirmando ser um desvio com uma previsão incorreta, um store ou qualquer outra instrução (confirmação normal). O caso da confirmação normal ocorre quando uma instrução alcança o início do ROB e seu resultado está presente no buffer; nesse ponto, o processador atualiza o registrador com o resultado e remove a instrução do ROB. A confirmação de um store é semelhante, exceto que a memória é atualizada, em vez de um registrador de resultado. Quando um desvio com previsão incorreta atinge o início do ROB, isso indica que a especulação foi errada. O ROB é esvaziado e a execução é reiniciada no sucessor correto do desvio. Se o desvio foi previsto corretamente, ele será terminado.

Quando uma instrução é confirmada, sua entrada no ROB é reclamada, e o registrador ou destino da memória é atualizado, eliminando a necessidade da entrada do ROB. Se o ROB se encher, simplesmente paramos de enviar instruções até que haja uma entrada

disponível. Agora, vamos examinar como esse esquema funcionaria com o mesmo exemplo que usamos para o algoritmo de Tomasulo.

Exemplo Vamos considerar as mesmas latências para as unidades funcionais de ponto flutuante que nos exemplos anteriores: adição usa dois ciclos de clock, multiplicação usa seis ciclos de clock e divisão usa 12 ciclos de clock. Usando o segmento de código a seguir, o mesmo que usamos para gerar a Figura 3.8, mostre como ficariam as tabelas de status quando o MUL.D estiver pronto para a confirmação.

```
L.D      F6,32(R2)
L.D      F2,44(R3)
MUL.D    F0,F2,F4
SUB.D    F8,F2,F6
DIV.D    F10,F0,F6
ADD.D    F6,F8,F2
```

Resposta A Figura 3.12 mostra o resultado nas três tabelas. Observe que, embora a instrução SUB.D tenha completado sua execução, ela não é confirmada até que o MUL.D seja confirmado. As estações de reserva e o campo de status do registrador contêm a mesma informação básica que eles tinham no algoritmo de Tomasulo (ver descrição desses campos na página 152). A diferença é que os números de estação de reserva são substituídos por números de entrada ROB nos campos Qj e Qk, e também nos campos de status de registrador, e acrescentamos o campo Dest às estações de reserva. O campo Dest designa a entrada do ROB, que é o destino para o resultado produzido por essa entrada da estação de reserva.

O exemplo ilustra a importante diferença-chave entre um processador com especulação e um processador com escalonamento dinâmico. Compare o conteúdo da Figura 3.12 com o da Figura 3.8, na página 154, que mostra a mesma sequência de código em operação em um processador com o algoritmo de Tomasulo. A principal diferença é que, no exemplo anterior, nenhuma instrução após a instrução mais antiga não completada (MUL.D acima) tem permissão para concluir. Ao contrário, na Figura 3.8, as instruções SUB.D e ADD.D também foram concluídas.

Uma implicação dessa diferença é que o processador com o ROB pode executar código dinamicamente enquanto mantém um modelo de interrupção preciso. Por exemplo, se a instrução MUL.D causasse uma interrupção, poderíamos simplesmente esperar até que ela atingisse o início do ROB e apanhar a interrupção, esvaziando quaisquer outras instruções pendentes do ROB. Como a confirmação da instrução acontece em ordem, isso gera uma exceção precisa.

Ao contrário, no exemplo usando o algoritmo de Tomasulo, as instruções SUB.D e ADD.D poderiam ser concluídas antes que o MUL.D levantasse a exceção. O resultado é que os registradores F8 e F6 (destinos das instruções SUB.D e ADD.D) poderiam ser sobrescritos e a interrupção seria imprecisa.

Alguns usuários e arquitetos decidiram que as exceções de ponto flutuante imprecisas são aceitáveis nos processadores de alto desempenho, pois o programa provavelmente terminará; veja no Apêndice J uma discussão aprofundada desse assunto. Outros tipos de exceção, como falhas de página, são muito mais difíceis de acomodar quando são imprecisas, pois o programa precisa retomar a execução transparentemente depois de tratar de tal exceção.

O uso de um ROB com a confirmação de instrução em ordem oferece exceções precisas, além de dar suporte à exceção especulativa, como mostra o exemplo seguinte.

Entrada	Busy	Instrução		Status	Destino	Valor
	Buffer de reordenação					
1	Não	`L.D`	`F6,32(R2)`	Confirmação	`F6`	`Mem[32 + Regs[R2]]`
2	Não	`L.D`	`F2,44(R3)`	Confirmação	`F2`	`Mem[44 + Regs[R3]]`
3	Sim	`MUL.D`	`F0,F2,F4`	Escrita de resultado	`F0`	#2 × `Regs[F4]`
4	Sim	`SUB.D`	`F8,F2,F6`	Escrita de resultado	`F8`	#2 – #1
5	Sim	`DIV.D`	`F10,F0,F6`	Execução	`F10`	
6	Sim	`ADD.D`	`F6,F8,F2`	Escrita de resultado	`F6`	#4 + #2

Nome	Busy	Op	Vj	Vk	Qj	Qk	Dest	A
	Estações de reserva							
Load1	Não							
Load2	Não							
Add1	Não							
Add2	Não							
Add3	Não							
Mult1	Não	MUL.D	`Mem[44 + Regs[R3]]`	`Regs[F4]`			#3	
Mult2	Sim	DIV.D		`Mem[32 + Regs[R2]]`	#3		#5	

Campo	F0	F1	F2	F3	F4	F5	F6	F7	F8	F10
	Status do registrador de PF									
Reord.#	3						6		4	5
Busy	Sim	Não	Não	Não	Não	Não	Sim	...	Sim	Sim

FIGURA 3.12 No momento em que o MUL.D está pronto para ser confirmado, somente as duas instruções L.D foram confirmadas, embora várias outras tenham completado sua execução.

O MUL.D está no início do ROB, e as duas instruções L.D estão lá somente para facilitar a compreensão. As instruções SUB.D e ADD.D não serão confirmadas até que a instrução MUL.D seja confirmada, embora os resultados das instruções estejam disponíveis e possam ser usados como fontes para outras instruções. O DIV.D está em execução, mas ainda não concluiu unicamente devido à sua latência maior do que MUL.D. A coluna Valor indica o valor sendo mantido; o formato #X é usado para se referir a um campo de valor da entrada X do ROB. Os buffers de reordenação 1 e 2 estão realmente concluídos, mas aparecem para fins informativos. Não mostramos as entradas para a fila load-store, mas essas entradas são mantidas em ordem.

Exemplo Considere o exemplo de código utilizado para o algoritmo de Tomasulo e mostrado na Figura 3.10 em execução:

```
Loop:   L.D      F0,0(R1)
        MUL.D    F4,F0,F2
        S.D      F4,0(R1)
        DADDIU   R1,R1,#-8
        BNE      R1,R2,Loop    ;branches if R1¦R2
```

Considere que tenhamos enviado todas as instruções no loop duas vezes. Vamos também considerar que o L.D e o MUL.D da primeira iteração foram confirmados e todas as outras instruções terminaram a execução. Normalmente, o store esperaria no ROB pelo operando de endereço efetivo (R1 neste exemplo) e pelo valor (F4 neste exemplo). Como só estamos considerando o pipeline de ponto flutuante, suponha que o endereço efetivo para o store seja calculado no momento em que a instrução é enviada.

Resposta A Figura 3.13 mostra o resultado em duas tabelas.

Como nem os valores de registradores nem quaisquer valores de memória são realmente escritos até que uma instrução seja confirmada, o processador poderá facilmente desfazer suas ações especulativas quando um desvio for considerado mal previsto. Suponha que o desvio BNE não seja tomado pela primeira vez na Figura 3.13. As instruções antes do desvio simplesmente serão confirmadas quando cada uma alcançar o início do ROB; quando o desvio alcançar o início desse buffer, o buffer será simplesmente apagado e o processador começará a apanhar instruções do outro caminho.

Na prática, os processadores que especulam tentam se recuperar o mais cedo possível após um desvio ser mal previsto. Essa recuperação pode ser feita limpando-se o ROB para todas as entradas que aparecem após o desvio mal previsto, permitindo que aquelas que estão antes do desvio no ROB continuem, reiniciando a busca no sucesso correto do desvio. Nos processadores especulativos, o desempenho é mais sensível à previsão do desvio, pois o impacto de um erro de previsão é mais alto. Assim, todos os aspectos do tratamento de desvios — exatidão da previsão, latência da detecção de erro de previsão e tempo de recuperação do erro de previsão — passam a ter mais importância.

As exceções são tratadas pelo seu não reconhecimento até que estejam prontas para serem confirmadas. Se uma instrução especulada levantar uma exceção, a exceção será registrada no ROB. Se um erro de previsão de desvio surgir e a instrução não tiver sido executada, a exceção será esvaziada junto com a instrução quando o ROB for apagado. Se a instrução atingir o início do ROB, saberemos que ela não é mais especulativa, e a exceção deverá realmente ser tomada. Também poderemos tentar tratar das exceções assim que elas surgirem e todos os desvios anteriores forem resolvidos, porém isso é mais desafiador no caso das exceções do que para o erro de previsão de desvio, pois ocorre com menos frequência e não é tão crítico.

	Buffer de reordenação					
Entrada	**Busy**	**Instrução**		**Status**	**Destino**	**Valor**
1	Não	L.D	F0,0(R1)	Confirmação	F0	Mem[0 +Regs[R1]]
2	Não	MUL.D	F4,F0,F2	Confirmação	F4	#1 × Regs[F2]
3	Sim	S.D	F4,0(R1)	Escrita de resultado	0 + Registros [R1]	#2
4	Sim	DADDIU	R1,R1,#-8	Escrita de resultado	R1	Regs[R1] – 8
5	Sim	BNE	R1,R2, Loop	Escrita de resultado		
6	Sim	L.D	F0,0(R1)	Escrita de resultado	F0	Mem[#4]
7	Sim	MUL.D	F4,F0,F2	Escrita de resultado	F4	#6 × Regs[F2]
8	Sim	S.D	F4,0(R1)	Escrita de resultado	0 + #4	#7
9	Sim	DADDIU	R1,R1,#-8	Escrita de resultado	R1	#4 – 8
10	Sim	BNE	R1,R2, Loop	Escrita de resultado		

	Status do registrador de PF								
Campo	**F0**	**F1**	**F2**	**F3**	**F4**	**F5**	**F6**	**F7**	**F8**
Reord.#	6				7				
Busy	Sim	Não	Não	Não	Sim	Não	Não	...	Não

FIGURA 3.13 Somente as instruções L.D e MUL.D foram confirmadas, embora todas as outras tenham a execução concluída.
Logo, nenhuma estação de reserva está ocupada e nenhuma aparece. As instruções restantes serão confirmadas o mais rápido possível. Os dois primeiros buffers de reordenação serão confirmados o mais rápido possível. Os dois primeiros buffers de reordenação estão vazios, mas aparecem para completar a figura.

A Figura 3.14 mostra as etapas da execução para uma instrução, além das condições que devem ser satisfeitas a fim de prosseguir para a etapa e as ações tomadas. Mostramos o caso em que os desvios mal previstos não são resolvidos antes da confirmação. Embora a especulação pareça ser um acréscimo simples ao escalonamento dinâmico, uma comparação da Figura 3.14 com a figura comparável para o algoritmo de Tomasulo na Figura 3.9 mostra que a especulação acrescenta complicações significativas ao controle. Além disso, lembre-se de que os erros de previsão de desvio também são um pouco mais complexos.

Existe uma diferença importante no modo como os stores são tratados em um processador especulativo e no algoritmo de Tomasulo. No algoritmo de Tomasulo, um store pode atualizar a memória quando alcançar a escrita de resultado (o que garante que o endereço efetivo foi calculado) e o valor de dados a armazenar estiver disponível. Em um processador especulativo, um store só atualiza a memória quando alcança o início do ROB. Essa diferença garante que a memória não seja atualizada até que uma instrução não seja mais especulativa.

A Figura 3.14 apresenta uma simplificação significativa para stores, que é desnecessária na prática. Ela exige que os stores esperem no estágio de escrita de resultado pelo registrador operando-fonte cujo valor deve ser armazenado; o valor é, então, movido do campo Vk da estação de reserva do store para o campo Valor da entrada de store do ROB. Porém, na realidade, o valor a ser armazenado não precisa chegar até *imediatamente antes* do store ser confirmado, e pode ser colocado diretamente na entrada de store do ROB, pela instrução de origem. Isso é realizado fazendo com que o hardware acompanhe quando o valor de origem a ser armazenado estará disponível na entrada de store do ROB e pesquisando o ROB a cada término de instrução para procurar stores dependentes.

Esse acréscimo não é complicado, mas sua inclusão tem dois efeitos: precisaríamos acrescentar um campo no ROB, e a Figura 3.14, que já está com uma fonte pequena, seria ainda maior! Embora a Figura 3.14 faça essa simplificação, em nossos exemplos permitiremos que o store passe pelo estágio de escrita de resultado e simplesmente espere que o valor esteja pronto quando for confirmado.

Assim como o algoritmo de Tomasulo, temos de evitar hazards na memória. Hazards WAW e WAR na memória são eliminados com a especulação, pois a atualização real da memória ocorre em ordem, quando um store está no início do ROB e, portanto, nenhum load ou store anterior poderá estar pendente. Os hazards RAW na memória são mantidos por duas restrições:

1. Não permitir que um load inicie a segunda etapa de sua execução se qualquer entrada ROB ativa ocupada por um store tiver um campo Destino que corresponda ao valor do campo A do load e
2. Manter a ordem do programa para o cálculo de endereço efetivo de um load com relação a todos os stores anteriores.

Juntas, essas duas restrições garantem que nenhum load que acesse um local da memória escrito por um store anterior poderá realizar o acesso à memória até que o store tenha escrito os dados. Alguns processadores especulativos realmente contornarão o valor do store para o load diretamente, quando ocorrer esse hazard RAW. Outra técnica é prever colisões em potencial usando uma forma de previsão de valor; consideraremos isso na Seção 3.9.

Embora essa explicação da execução especulativa tenha focalizado o ponto flutuante, as técnicas se estendem facilmente para registradores inteiros e unidades funcionais, conforme veremos na seção "Juntando tudo". Na realidade, a especulação pode ser mais útil em programas de inteiros, pois esses programas costumam ter um código em que o comportamento do desvio é menos previsível. Além disso, essas técnicas podem ser estendidas para que funcionem em um processador de múltiplo despacho, permitindo que várias instruções

Status	Esperar até	Ação ou manutenção

Enviar todas as instruções	Estação de reserva (r) e ROB(b) disponíveis	if (RegisterStat[rs].Busy)/*instr. no ato escreve rs*/
		{h←RegisterStat[rs].Reorder;
		if (ROB[h].Ready)/* Instr. já completada */
		{RS[r].Vj←ROB[h].Value; RS[r].Qj←0;}
		else {RS[r].Qj←h;} /* espera instrução */
		} else {RS[r].Vj←Regs[rs]; RS[r].Qj←0;};
		RS[r].Busy←yes; RS[r].Dest←b;
		ROB[b].Instruction←opcode; ROB[b].Dest←rd;ROB[b].Ready←no;

Operações de PF e stores		if (RegisterStat[rt].Busy) /*instr. no ato escreve rt*/
		{h←RegisterStat[rt].Reorder;
		if (ROB[h].Ready)/* Instr. já completada */
		{RS[r].Vk←ROB[h].Value; RS[r].Qk←0;}
		else {RS[r].Qk←h;} /* espera instrução */
		} else {RS[r].Vk←Regs[rt]; RS[r].Qk←0;};

Operações de PF		RegisterStat[rd].Reorder←b; RegisterStat[rd].Busy←yes; ROB[b].Dest←rd;
Loads		RS[r].Aimm; RegisterStat[rt].Reorder←b; RegisterStat[rt].Busy←yes; ROB[b].Dest←rt;
Stores		RS[r].A←imm;
Executar op. de PF	(RS[r].Qj = = 0) e (RS[r].Qk = = 0)	Calcular resultados – operandos estão em Vj e Vk
Load etapa 1	(RS[r].Qj = = 0) e não existem stores anteriormente na fila	RS[r].A←RS[r].Vj + RS[r].A;
Load etapa 2	Load etapa 1 feita e todos os stores anteriores no ROB têm endereço diferente	Ler de Mem[RS[r].A]
Store	(RS[r].Qj = = 0) e store no	ROB[h].Address←RS[r].Vj + RS[r].A;
Escrita de resultado, sem o store	Execução feita em *r* e CDB disponível	b←RS[r].Dest; RS[r].Busy←no; ∀x(if (RS[x].Qj= =b) {RS[x].Vj←result; RS[x].Qj←0}); ∀x(if (RS[x].Qk= =b) {RS[x].Vk←result; RS[x].Qk←0}); ROB[b].Value← result; ROB[b].Ready←yes;
Store	Execução feita em *r* e (RS[r].Qk = = 0)	ROB[h].Value←RS[r].Vk;

Confirmação	Instrução está no início do ROB (entrada *h*) e ROB[h].ready = = yes	d←ROB[h].Dest; /* dest registr, se houver*/
		if (ROB[h].Instruction= =Branch)
		{if (desvio mal previsto)
		{clear ROB[h], RegisterStat; fetch branch dest;};}
		else if (ROB[h].Instruction= =Store)
		{Mem[ROB[h].Destination] ←ROB[h].Value;}
		else /* coloca resultado no destino do registrador */
		{Regs[d]←ROB[h].Value;};
		ROB[h].Busy←no; /* libera entrada do ROB */
		/* libera reg. destino se ninguém mais estiver escrevendo */
		if (RegisterStat[d] = .Reorder= =h) {RegisterStat[d] = .Busy←no;};

FIGURA 3.14 **Etapas no algoritmo e o que é necessário para cada etapa.**
Para as instruções enviadas, rd é o destino, rs e rt são os fontes, *r* é a estação de reserva alocada, *b* é a entrada do ROB atribuída e *h* é a entrada inicial do ROB. RS é a estrutura de dados da estação de reserva. O valor retornado por uma estação de reserva é chamado de resultado. RegisterStat é a estrutura de dados do registrador, Regs representa os registradores reais e ROB é a estrutura de dados do buffer de reordenação.

sejam enviadas e confirmadas a cada clock. Na verdade, a especulação provavelmente é mais interessante nesses processadores, pois talvez técnicas menos ambiciosas possam explorar um ILP suficiente dentro dos blocos básicos quando auxiliado por um compilador.

3.7 EXPLORANDO O ILP COM MÚLTIPLO DESPACHO E ESCALONAMENTO ESTÁTICO

As técnicas das seções anteriores podem ser usadas para eliminar stalls de dados e controle e alcançar um CPI ideal de 1. Para melhorar ainda mais o desempenho, gostaríamos de diminuir o CPI para menos de 1. Mas o CPI não pode ser reduzido para menos de 1 se enviarmos apenas uma instrução a cada ciclo de clock.

O objetivo dos *processadores de múltiplo despacho*, discutidos nas próximas seções, é permitir que múltiplas instruções sejam enviadas em um ciclo de clock. Os processadores de múltiplo despacho podem ser de três tipos principais:

1. Processadores superescalares escalonados estaticamente.
2. Processadores VLIW (Very Long Instruction Word).
3. Processadores superescalares escalonados dinamicamente.

Os dois tipos de processadores superescalares enviam números variados de instruções por clock e usam a execução em ordem quando são estaticamente escalonados ou a execução fora da ordem quando são dinamicamente escalonados.

Processadores VLIW, ao contrário, enviam um número fixo de instruções formatadas como uma instrução grande ou como um pacote de instrução fixo com o paralelismo entre instruções indicado explicitamente pela instrução. Processadores VLIW são inerentemente escalonados de forma estática pelo compilador. Quando a Intel e a HP criaram a arquitetura IA-64, descrita no Apêndice H, também introduziram o nome EPIC (Explicitly Parallel Instruction Computer) para esse estilo de arquitetura.

Embora os superescalares escalonados estaticamente enviem um número variável e não um número fixo de instruções por clock, na verdade, em conceito, eles estão mais próximos aos VLIWs, pois as duas técnicas contam com o compilador para escalonar o código para o processador. Devido às vantagens cada vez menores de um superescalar escalonado estaticamente à medida que a largura de despacho aumenta, os superescalares escalonados estaticamente são usados principalmente para larguras de despacho estreitas, geralmente apenas com duas instruções. Além dessa largura, a maioria dos projetistas escolhe implementar um VLIW ou um superescalar escalonado dinamicamente. Devido às semelhanças no hardware e tecnologia de compilador exigida, nesta seção enfocamos os VLIWs. Os conhecimentos desta seção são facilmente extrapolados para um superescalar escalonado estaticamente.

A Figura 3.15 resume as técnicas básicas para o múltiplo despacho e suas características distintas, mostrando os processadores que usam cada técnica.

A técnica VLIW básica

Os VLIWs utilizam múltiplas unidades funcionais independentes. Em vez de tentar enviar múltiplas instruções independentes para as unidades, um VLIW empacota as múltiplas operações em uma instrução longa ou exige que as instruções no pacote de despacho satisfaçam as mesmas restrições. Como não existe diferença fundamental nas duas técnicas, assumiremos apenas que múltiplas operações são colocadas em uma instrução, como na técnica VLIW original.

Como a vantagem de um VLIW aumenta à medida que a taxa de despacho máxima cresce, enfocamos um processador com largura de despacho maior. Na realidade, para proces-

Nome comum	Estrutura de despacho	Detecção de hazard	Escalonamento	Característica de distinção	Exemplos
Superescalar (estática)	Dinâmica	Hardware	Estático	Execução em ordem	Em sua maioria no espaço embarcado: MIPS e ARM, incluindo o ARM Cotex-A8
Superescalar (dinâmica)	Dinâmica	Hardware	Dinâmico	Alguma execução fora de ordem, mas sem especulação	Nenhum atualmente
Superescalar (especulativa)	Dinâmica	Hardware	Dinâmico com especulação	Execução fora de ordem com especulação	Pentium 4, MIPS R12K, IBM Power5
VLIW/LIW	Estática	Principalmente software	Estático	Todos os hazards determinados e indicados pelo compilador (geralmente implícito)	A maioria dos exemplos está nos processadores de sinais digitais, como o TI C6x
EPIC	Principalmente estática	Principalmente software	A maior parte estático	Todos os hazards determinados e indicados explicitamente pelo compilador	Itanium

FIGURA 3.15 As cinco técnicas principais em uso para processadores de múltiplo despacho e as principais características que os distinguem. Este capítulo enfoca as técnicas com uso intensivo de hardware, que são todas de alguma forma de superescalar. O Apêndice G enfoca as técnicas baseadas em compilador. A técnica EPIC, incorporada à arquitetura IA-64, estende muitos dos conceitos das primeiras técnicas VLIW, oferecendo uma mistura de técnicas estáticas e dinâmicas.

sadores simples de despacho com largura dois, o overhead de um superescalar provavelmente é mínimo. Muitos projetistas provavelmente argumentariam que um processador de despacho quádruplo possui overhead controlável, mas, como veremos mais adiante neste capítulo, o crescimento no overhead é um fator importante que limita os processadores com larguras de despacho maiores.

Vamos considerar um processador VLIW com instruções que contêm cinco operações, incluindo uma operação com inteiros (que também poderia ser um desvio), duas operações de ponto flutuante e duas referências à memória. A instrução teria um conjunto de campos para cada unidade funcional — talvez 16-24 bits por unidade, gerando um tamanho de instrução de algo entre 80-120 bits. Por comparação, o Intel Itanium 1 e o 2 contêm seis operações de instruções por pacote (ou seja, eles permitem o despacho concorrente de dois conjuntos de três instruções, como descreve o Apêndice H).

Para manter as unidades funcionais ocupadas, é preciso haver paralelismo suficiente em uma sequência de código para preencher os slots de operação disponíveis. Esse paralelismo é descoberto desdobrando os loops e escalonando o código dentro do único corpo de loop maior. Se o desdobramento gerar código sem desvios ou loops (straight-line code), as técnicas de *escalonamento local*, que operam sobre um único bloco básico, podem ser utilizadas. Se a localização e a exploração do paralelismo exigirem escalonamento de código entre os desvios, um algoritmo de *escalonamento global* substancialmente mais complexo terá de ser usado. Os algoritmos de escalonamento global não são apenas mais complexos em estrutura, mas também precisam lidar com escolhas significativamente mais complicadas em otimização, pois a movimentação de código entre os desvios é dispendiosa.

No Apêndice H, discutiremos o *escalonamento de rastreio*, uma dessas técnicas de escalonamento global desenvolvidas especificamente para VLIWs; também exploraremos o suporte especial de hardware, que permite que alguns desvios condicionais sejam eliminados, estendendo a utilidade do escalonamento local e melhorando o desempenho do escalonamento global.

Por enquanto, contaremos com o desdobramento do loop para gerar sequências de código longas, straight-line, a fim de podermos usar o escalonamento local para montar instruções VLIW e explicar o quanto esses processadores operam bem.

Exemplo Suponha que tenhamos um VLIW que possa enviar duas referências à memória, duas operações de PF e uma operação com inteiros ou desvio a cada ciclo de clock. Mostre uma versão desdobrada do loop x[i] = x[i] + s (ver código MIPS na página 136) para tal processador. Desdobre tantas vezes quantas forem necessárias para eliminar quaisquer stalls. Ignore os desvios adiados.

Resposta A Figura 3.16 mostra o código. O loop foi desdobrado para fazer sete cópias do corpo, o que elimina quaisquer stalls (ou seja, ciclos de despacho completamente vazios), sendo executado em nove ciclos. Esse código gera uma taxa de execução de sete resultados em nove ciclos ou 1,29 ciclo por resultado, quase o dobro da rapidez do superescalar de despacho duplo da Seção 3.2, que usava código desdobrado e escalonado.

Para o modelo VLIW original, havia problemas técnicos e logísticos que tornavam a técnica menos eficiente. Os problemas técnicos são o aumento no tamanho do código e as limitações da operação de bloqueio. Dois elementos diferentes são combinados para aumentar o tamanho do código substancialmente para um VLIW. Primeiro, a geração de operações suficientes em um fragmento de código straight-line requer desdobramento de loops de ambiciosos (como nos exemplos anteriores), aumentando assim o tamanho do código. Segundo, sempre que as instruções não forem cheias, as unidades funcionais não usadas se traduzem em bits desperdiçados na codificação de instrução. No Apêndice H, examinamos as técnicas de escalonamento de software, como o pipelining de software, que podem alcançar os benefícios do desdobramento sem muita expansão do código.

Para combater esse aumento no tamanho do código, às vezes são utilizadas codificações inteligentes. Por exemplo, pode haver apenas um campo imediato grande para uso por qualquer unidade funcional. Outra técnica é compactar as instruções na memória principal e expandi-las quando forem lidas para a cache ou quando forem decodificadas. No Apêndice H, mostramos outras técnicas, além de documentar a significativa expansão do código vista no IA-64.

Referência de memória 1	Referência de memória 2	Operação de PF 1	Operação de PF 2	Operação de inteiros/desvio
L.D F0,0(R1)	L.D F6,-8(R1)			
L.D F10,-16(R1)	L.D F14,-24(R1)			
L.D F18,-32(R1)	L.D F22,-40(R1)	ADD.D F4,F0,F2	ADD.D F8,F6,F2	
L.D F26,-48(R1)		ADD.D F12,F10,F2	ADD.D F16,F14,F2	
		ADD.D F20,F18,F2	ADD.D F24,F22,F2	
S.D F4,0(R1)	S.D F8,-8(R1)	ADD.D F28,F26,F2		
S.D F12,-16(R1)	S.D F16,-24(R1)			DADDUI R1,R1,#-56
S.D F20,24(R1)	S.D F24,16(R1)			
S.D F28,8(R1)				BNE R1,R2,Loop

FIGURA 3.16 **Instruções VLIW que ocupam o loop interno e substituem a sequência desdobrada.**
Esse código usa nove ciclos, considerando que não haja atraso de desvio; normalmente, o atraso de desvio também precisaria ser escalonado. A taxa de despacho é de 23 operações em nove ciclos de clock ou 2,5 operações por ciclo. A eficiência, ou a porcentagem de slots disponíveis que continuam uma operação, é de cerca de 60%. Para conseguir essa taxa de despacho, é preciso um número maior de registradores do que o MIPS normalmente usaria nesse loop. A sequência de código VLIW acima requer pelo menos oito registradores FP, enquanto a mesma sequência de código para o processador MIPS básico pode usar desde dois até cinco registradores de PF, quando desdobrada e escalonada.

Os primeiros VLIWs operavam em bloqueio; não havia hardware de detecção de hazard algum. Essa estrutura ditava que um stall em qualquer pipeline de unidade funcional precisa fazer com que o processador inteiro pare e espere, pois todas as unidades funcionais precisam ser mantidas em sincronismo. Embora um compilador possa ser capaz de escalonar as unidades funcionais determinísticas para evitar os stalls, é muito difícil prever quais acessos aos dados causarão um stall de cache e escaloná-los. Logo, as caches precisavam de bloqueio, fazendo com que *todas* as unidades funcionais protelassem. À medida que a taxa de despacho e o número de referências à memória se tornava grande, essa restrição de sincronismo se tornou inaceitável. Em processadores mais recentes, as unidades funcionais operam de forma mais independente, e o compilador é usado para evitar hazards no momento do despacho, enquanto as verificações de hardware permitem a execução não sincronizada quando as instruções são enviadas.

A compatibilidade do código binário também tem sido um problema logístico importante para os VLIWs. Em uma técnica VLIW estrita, a sequência de código utiliza a definição do conjunto de instruções e a estrutura de pipeline detalhada, incluindo as unidades funcionais e suas latências. Assim, diferentes quantidades de unidades funcionais e latências de unidade exigem diferentes versões do código. Esse requisito torna a migração entre implementações sucessivas, ou entre implementações com diferentes larguras de despacho, mais difícil do que para um projeto superescalar. Naturalmente, a obtenção de desempenho melhorado a partir de um novo projeto de superescalar pode exigir recompilação. Apesar disso, a capacidade de executar arquivos binários antigos é uma vantagem prática para uma técnica superescalar.

A técnica EPIC, da qual a arquitetura IA-64 é o principal exemplo, oferece soluções para muitos dos problemas encontrados nos primeiros projetos VLIW, incluindo extensões para uma especulação de software mais agressiva e métodos para contornar a limitação da dependência do hardware enquanto preserva a compatibilidade binária.

O principal desafio para todos os processadores de múltiplo despacho é tentar explorar grande quantidade de ILP. Quando o paralelismo vem do desdobramento de loops simples em programas de PF, provavelmente o loop original não foi executado de forma eficiente em um processador vetorial (descrito no Capítulo 4). Não é claro que um processador de múltiplo despacho seja preferido em relação a um processador vetorial para tais aplicações; os custos são semelhantes, e o processador vetorial normalmente tem a mesma velocidade ou é mais rápido. As vantagens em potencial de um processador de múltiplo despacho *versus* um processador vetorial são sua capacidade de extrair algum paralelismo do código menos estruturado e sua capacidade de facilmente colocar em cache todas as formas de dados. Por esses motivos, as técnicas de múltiplo despacho se tornaram o método principal para tirar proveito do paralelismo em nível de instrução, e os vetores se tornaram principalmente uma extensão desses processadores.

3.8 EXPLORANDO O ILP COM ESCALONAMENTO DINÂMICO, MÚLTIPLO DESPACHO E ESPECULAÇÃO

Até aqui, vimos como funcionam os mecanismos individuais do escalonamento dinâmico, múltiplo despacho e especulação. Nesta seção, juntamos os três, o que gera uma microarquitetura muito semelhante àquelas dos microprocessadores modernos. Para simplificar, consideramos apenas uma taxa de despacho de duas instruções por clock, mas os conceitos não são diferentes dos processadores modernos, que enviam três ou mais instruções por clock.

Vamos considerar que queremos estender o algoritmo de Tomasulo para dar suporte a um pipeline superescalar de despacho duplo, com uma unidade de inteiros e de ponto flutuante separadas, cada qual podendo iniciar uma operação a cada clock. Não queremos enviar instruções fora de ordem para as estações de reserva, pois isso levaria a uma violação da semântica do programa. Para tirar proveito total do escalonamento dinâmico, permitiremos que o pipeline envie qualquer combinação de duas instruções em um clock, usando o hardware de escalonamento para realmente atribuir operações à unidade de inteiros e à de ponto flutuante. Como a interação das instruções de inteiros e de ponto flutuante é crucial, também estendemos o esquema de Tomasulo para lidar com as unidades funcionais e os registradores de inteiros e de ponto flutuante, além de incorporar a execução especulativa. Como a Figura 3.17 mostra, a organização básica é similar àquela de um processador com especulação com um despacho por clock, exceto pelo fato de que a lógica de despacho e conclusão deve ser melhorada para permitir múltiplas instruções por clock.

Emitir instruções múltiplas por clock em um processador escalonado dinamicamente (com ou sem especulação) é muito complexo pela simples razão de que as múltiplas

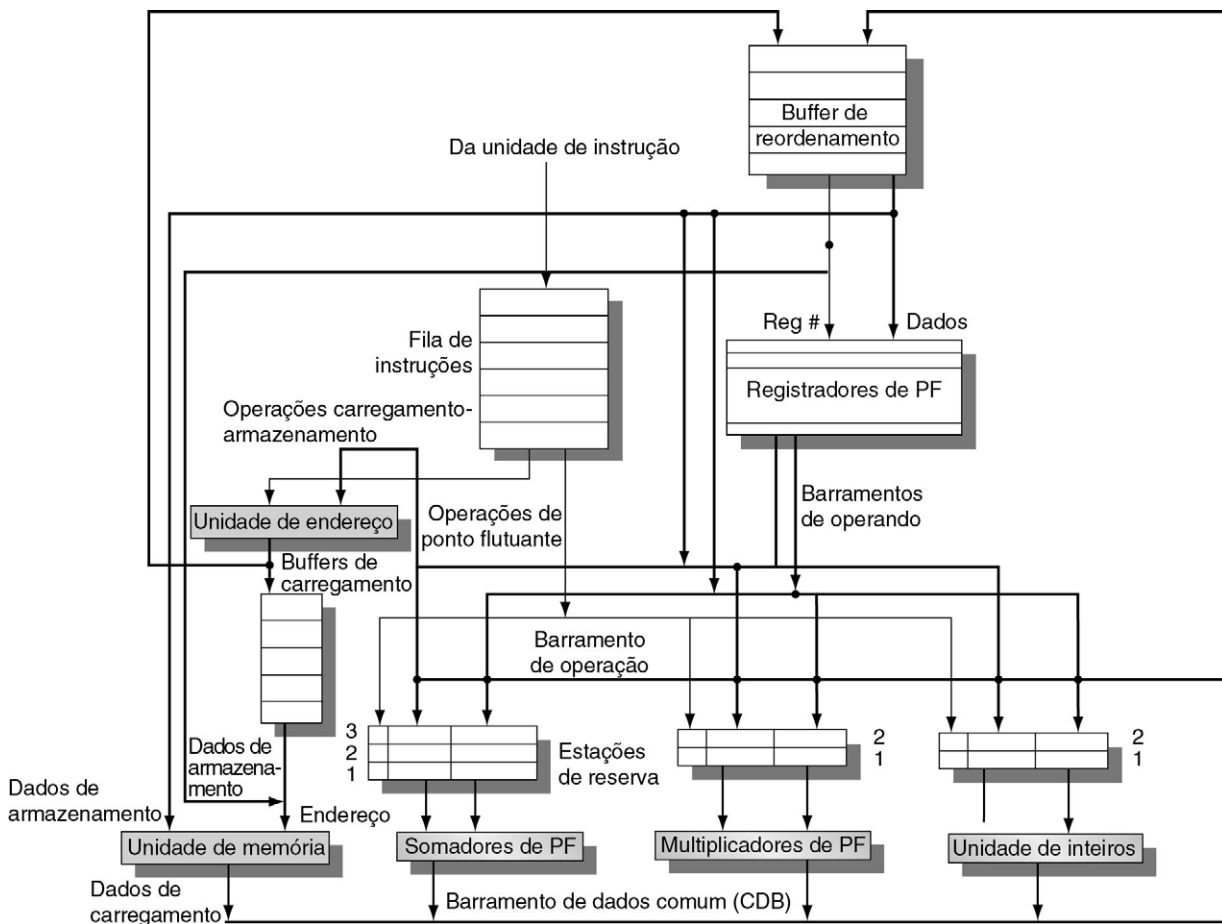



FIGURA 3.17 **A organização básica de um processador de múltiplos despachos com especulação.**
Neste caso, a organização poderia permitir uma multiplicação de PF, uma soma de PF, inteiros e load/store simultaneamente para todos os despachos (supondo um despacho por clock por unidade funcional). Observe que diversos caminhos de dados devem ser alargados para suportar múltiplos despachos: o CDB, os barramentos de operando e, essencialmente, a lógica de despacho de instrução, que não é mostrada nesta figura. O último é um problema difícil, como discutiremos no texto.

instruções podem depender umas das outras. Por isso, as tabelas devem ser atualizadas para as instruções em paralelo. Caso contrário, ficarão incorretas ou a dependência poderá ser perdida.

Duas técnicas diferentes foram usadas para enviar múltiplas instruções por clock em um processador escalonado dinamicamente, e ambas contam com a observação de que a chave é atribuir uma estação de reserva e atualizar as tabelas de controle de pipeline. Uma técnica é executar essa etapa na metade de um ciclo de clock, de modo que duas instruções possam ser processadas em um ciclo de clock. Infelizmente, essa técnica não pode ser estendida facilmente para lidar com quatro instruções por clock.

Uma segunda alternativa é montar a lógica necessária para lidar com duas instruções ao mesmo tempo, incluindo quaisquer dependências possíveis entre as instruções. Os modernos processadores superescalares que enviam quatro ou mais instruções por clock normalmente incluem ambas as técnicas: ambas utilizam uma lógica de despacho de largura grande e com o pipeline. Uma observação importante é que não podemos eliminar o problema com pipelining. Ao fazer com que os despachos de instrução levem múltiplos clocks, porque novas instruções são enviadas a cada ciclo de clock, devemos ser capazes de definir a estação de reserva e atualizar as tabelas de pipeline, de modo que uma instrução dependente enviada no próximo clock possa usar as informações atualizadas.

Esse passo de despacho é um dos funis mais fundamentais em superscalares escalonados dinamicamente. Para ilustrar a complexidade desse processo, a Figura 3.18 mostra a lógica de despacho para um caso: enviar um load seguido por uma operação FP dependente. A lógica se baseia na da Figura 3.14, na página 166, mas representa somente um caso. Em um superescalar moderno, todas as combinações possíveis de instruções dependentes que se podem enviar em um clock, o passo do despacho é um gargalo provável para tentativas de ir além de quatro instruções por clock.

Podemos generalizar os detalhes da Figura 3.18 para descrever a estratégia básica para atualizar a lógica de despacho e as tabelas de reserva em um superescalar escalonado dinamicamente com até *n* despachos por clock como a seguir:

1. Definir uma estação de reserva e um buffer de reordenação para *todas* as instruções que *podem* ser enviadas no próximo conjunto de despacho. Essa definição pode ser feita antes que os tipos de instruções sejam conhecidos, simplesmente pré-alocando as entradas do buffer de reordenação e garantindo que estejam disponíveis estações de reserva o suficiente para enviar todo o conjunto, independentemente do que ele contenha. Ao limitar o número de instruções de uma dada classe (digamos, um FP, um inteiro, um load, um store), as estações de reserva necessárias podem ser pré-alocadas. Se estações de reserva suficientes não estiverem disponíveis (como quando as próximas instruções do programa são todas do mesmo tipo), o conjunto será quebrado, e somente um subconjunto das instruções, na ordem do programa original, será enviado. O restante das instruções no conjunto poderá ser colocado no próximo conjunto para despacho em potencial.
2. Analisar todas as dependências entre as instruções no conjunto de despacho.
3. Se uma instrução no conjunto depender de uma instrução anterior no conjunto, use o número do buffer de reorganização atribuído para atualizar a tabela de reserva para a instrução dependente. Caso contrário, use a tabela de reserva existente e a informação do buffer de reorganização para atualizar as entradas da tabela de reservas para as instruções enviadas.

Obviamente, o que torna isso muito complicado é o fato de que tudo é feito em paralelo em um único ciclo de clock!

Ação ou manutenção	Comentários

`if (RegisterStat[rs1].Busy)/*instr. no ato escreve rs*/`
`  {h←RegisterStat[rs1].Reorder;`
`  if (ROB[h].Ready)/* Instr. já completada */`
`    {RS[r1].Vj←ROB[h].Value; RS[r1].Qj←0;}`
`  else {RS[r1].Qj←h;} /* espera instrução */`
`} else {RS[r1].VjRegs[rs]; RS[r1].Qj0;};RS[r1].Busy←yes;`
`RS[r1].Dest←b`
`ROB[b1].Instruction ←Load; ROB[b1].Dest ←rd1;`
`ROB[b1].Ready ←no;`
`RS[r].A ←imm1; RegisterStat[rt1].Reorder ←b1;`
`RegisterStat[rt1].Busy ←yes; ROB[b1].Dest ←rt1;`
	Atualizar as tabelas de reserva para a instrução de load, que tem um único operando-fonte. Já que essa é a primeira instrução nesse conjunto de despachos, ela não parece diferente da que normalmente aconteceria em um load.

`RS[r2].Qj ← b1;} /* espera instrução de load*/`	Como o primeiro operando da operação de PF é do load, esse passo simplesmente atualiza a estação de reserva para apontar para o load. Observe que a dependência deve ser analisada durante o processamento e que as entradas ROB devem ser alocadas durante esta etapa do despacho para que as tabelas de reserva possam ser atualizadas corretamente.

`if (RegisterStat[rt2].Busy) /* instr. no ato escreve rt*/`
`  {h ← RegisterStat[rt2].Reorder;`
`  if (ROB[h].Ready)/* Instr. já completada */`
`    {RS[r2].Vk ←ROB[h].Value; RS[r2].Qk ←0;}`
`  else {RS[r2].Qk ←h;} /* espera instrução */`
`} else {RS[r2].Vk ←Regs[rt2]; RS[r2].Qk ←0;};`
`RegisterStat[rd2].Reorder ←b2;`
`RegisterStat[rd2].Busy ←yes;`
`ROB[b2].Dest←rd2;`
	Uma vez que supomos que o segundo operando da instrução de PF veio de um conjunto de despacho anterior, esse passo se parece com o modo como seria no caso de despacho único. É claro, se essa instrução fosse dependente de algo no mesmo conjunto de despacho, as tabelas precisariam ser atualizadas usando o buffer de reserva designado.

`RS[r2].Busy ←yes; RS[r2].Dest ←b2;`
`ROB[b2].Instruction ←FP operation; ROB[b2].Dest ←rd2;`
`ROB[b2].Ready ←no;`
	Essa seção simplesmente atualiza as tabelas para a operação de PF e é independente do load. Obviamente, se mais instruções nesse conjunto de instruções dependessem da operação de PF (como aconteceria com um superescalar de quatro despachos), as atualizações das tabelas de reserva para essas instruções seriam afetadas por essa instrução.

FIGURA 3.18 Passos de despacho para um par de instruções dependentes (chamadas 1 e 2), onde a instrução 1 é um load de PF e a instrução 2 é uma operação de PF cujo primeiro operando é o resultado da instrução de load; r1 e r2 são as estações de reserva designadas para as instruções; b1 e b2 são as entradas de buffer de reordenação designadas.

Para as instruções enviadas, rd1 e rd2 são os destinos, rs1, rs2 e rt2 são as fontes (o load tem somente uma fonte); r1 e r2 são as estações de reserva alocadas; b1 e *b* são as entradas ROB designadas. RS é a estrutura de dados da estação de reserva. RegisterStat é a estrutura de dados de registrador, Regs representa os registradores reais e ROB é a estrutura de dados do buffer de reorganização. Observe que precisamos ter entradas de buffer de reorganização designadas para que essa lógica opere corretamente e lembrar que todas essas atualizações ocorrem em um único ciclo de clock em paralelo, não sequencialmente!

Na extremidade final do pipeline, devemos ser capazes de completar e emitir múltiplas instruções por clock. Esses passos são um pouco mais fáceis do que os problemas de despacho, uma vez que múltiplas instruções que podem realmente ser emitidas no mesmo ciclo de clock já devem ter sido tratadas e quaisquer dependências resolvidas. Como veremos mais adiante, os projetistas descobriram como lidar com essa complexidade: o Intel i7, que examinaremos na Seção 3.13, usa essencialmente o esquema que descrevemos para múltiplos despachos especulativos, incluindo grande número de estações de reserva, um buffer de reorganização e um buffer de load e store, que também é usado para tratar faltas de cache sem bloqueio.

Do ponto de vista do desempenho, podemos mostrar como os conceitos se encaixam com um exemplo.

Exemplo Considere a execução do loop a seguir, que incrementa cada elemento de um array inteiro, em um processador de duplo despacho, uma vez sem especulação e uma com especulação:

```
Loop:   LD      R2,0(R1)     ;R2=array element
        DADDIU  R2,R2,#1     ;increment R2
        SD      R2,0(R1)     ;store result
        DADDIU  R1,R1,#8     ;increment pointer
        BNE     R2,R3,LOOP   ;branch if not last element
```

Suponha que existam unidades funcionais separadas de inteiros para cálculo eficaz de endereço, para operações de ALU e para avaliação de condição de desvios. Crie uma tabela para as três primeiras iterações desse loop para os dois processadores. Suponha que até duas instruções de qualquer tipo possam ser emitidas por clock.

Resposta As Figuras 3.19 e 3.20 mostram o desempenho para um processador escalonado dinamicamente com duplo despacho, sem e com especulação. Nesse caso, onde um desvio pode ser um limitador crítico do desempenho, a especulação ajuda significativamente. O terceiro desvio em que o processador é executado no ciclo de clock 13, enquanto no pipeline não especulativo, ele é executado no ciclo de clock 19. Já que a taxa de conclusão no pipeline não especulativo está ficando rapidamente para trás da taxa de despacho, o pipeline não especulativo vai sofrer stall quando algumas poucas iterações a mais forem enviadas. O desempenho do processador não especulativo poderia ser melhorado permitindo que as instruções load completassem o cálculo do endereço efetivo antes de um desvio ser decidido, mas sem que sejam permitidos acessos especulativos à memória, essa melhoria vai ganhar somente um clock por interação.

Número da iteração	Instruções		Despacha no ciclo de clock número	Executa no ciclo número	Acessa memória no ciclo número	Escreve no CDB no ciclo número	Comentário
1	LD	R2,0(R1)	1	2	3	4	Primeiro despacho
1	DADDIU	R2,R2,#1	1	5		6	Espera por LW
1	SD	R2,0(R1)	2	3	7		Espera por DADDIU
1	DADDIU	R1,R1,#8	2	3		4	Executa diretamente
1	BNE	R2,R3,LOOP	3	7			Espera por DADDIU
2	LD	R2,0(R1)	4	8	9	10	Espera por BNE
2	DADDIU	R2,R2,#1	4	11		12	Espera por LW
2	SD	R2,0(R1)	5	9	13		Espera por DADDIU
2	DADDIU	R1,R1,#8	5	8		9	Espera por BNE
2	BNE	R2,R3,LOOP	6	13			Espera por DADDIU
3	LD	R2,0(R1)	7	14	15	16	Espera por BNE
3	DADDIU	R2,R2,#1	7	17		18	Espera por LW
3	SD	R2,0(R1)	8	15	19		Espera por DADDIU
3	DADDIU	R1,R1,#8	8	14		15	Espera por BNE
3	BNE	R2,R3,LOOP	9	19			Espera por DADDIU

FIGURA 3.19 Tempo de despacho, execução e escrita de resultado para uma versão de despacho duplo do nosso pipeline sem especulação. Observe que o LD seguido pelo BNE não pode iniciar a execução antes, pois precisa esperar até que o resultado do desvio seja determinado. Esse tipo de programa, com desvios dependentes de dados que não podem ser resolvidos anteriormente, mostra a força da especulação. Unidades funcionais separadas para o cálculo do endereço, operações com a ALU e avaliação da condição de desvio permitem que várias instruções sejam executadas no mesmo ciclo. A Figura 3.20 mostra esse exemplo com especulação.

Número da iteração	Instruções		Ciclo de clock número	Executa no ciclo número	Acessa memória no ciclo número	Escreve no CDB no ciclo número	Confirma no ciclo #	Comentário
1	LD	R2,0(R1)	1	2	3	4	5	Primeiro despacho
1	DADDIU	R2,R2,#1	1	5		6	7	Espera por LW
1	SD	R2,0(R1)	2	3			7	Espera por DADDIU
1	DADDIU	R1,R1,#8	2	3		4	8	Confirma na ordem
1	BNE	R2,R3,LOOP	3	7			8	Espera por DADDIU
2	LD	R2,0(R1)	4	5	6	7	9	Espera sem execução
2	DADDIU	R2,R2,#1	4	8		9	10	Espera por LW
2	SD	R2,0(R1)	5	6			10	Espera por DADDIU
2	DADDIU	R1,R1,#8	5	6		7	11	Confirma na ordem
2	BNE	R2,R3,LOOP	6	10			11	Espera por DADDIU
3	LD	R2,0(R1)	7	8	9	10	12	O mais cedo possível
3	DADDIU	R2,R2,#1	7	11		12	13	Espera por LW
3	SD	R2,0(R1)	8	9			13	Espera por DADDIU
3	DADDIU	R1,R1,#8	8	9		10	14	Executa mais cedo
3	BNE	R2,R3,LOOP	9	13			14	Espera por DADDIU

FIGURA 3.20 **Tempo de despacho, execução e escrita de resultado para uma versão de despacho duplo de nossos pipelines com especulação.** Observe que o LD seguido do BNE pode iniciar a execução mais cedo, pois é especulativo.

Esse exemplo mostra claramente como a especulação pode ser vantajosa quando existem desvios dependentes dos dados, que de outro modo limitariam o desempenho. Entretanto, essa vantagem depende da previsão precisa de desvios. A especulação incorreta não melhora o desempenho. Na verdade, geralmente ela prejudica o desempenho e, como veremos, diminui drasticamente a eficiência energética.

3.9 TÉCNICAS AVANÇADAS PARA O DESPACHO DE INSTRUÇÕES E ESPECULAÇÃO

Em um pipeline de alto desempenho, especialmente um com múltiplo despacho, prever bem os desvios não é suficiente; na realidade, temos de ser capazes de entregar um fluxo de instrução com uma grande largura de banda. Nos processadores recentes de múltiplo despacho, isso significa resolver 4-8 instruções a cada ciclo de clock. Primeiro veremos os métodos para aumentar a largura de banda de despacho de instrução. Depois, passaremos a um conjunto de questões fundamentais na implementação de técnicas avançadas de especulação, incluindo o uso de renomeação de registrador *versus* buffers de reordenação, a agressividade da especulação e uma técnica chamada previsão de valor, que poderia melhorar ainda mais o ILP.

Aumentando a largura de banda da busca de instruções (instruction fetch)

Um processador de múltiplo despacho exigirá que o número médio de instruções buscadas a cada ciclo de clock seja pelo menos do mesmo tamanho do throughput médio. Naturalmente, buscar essas instruções exige caminhos largos o suficiente para a cache

da instrução, mas o aspecto mais difícil é lidar com desvios. Nesta seção, veremos dois métodos para lidar com desvios e depois discutiremos como os processadores modernos integram as funções de previsão e pré-fetch de instrução.

Buffers de destino de desvio

Para reduzir a penalidade do desvio para o nosso pipeline simples de cinco estágios, além dos pipelines mais profundos, temos de saber se a instrução ainda não decodificada é um desvio e, se for, qual deverá ser o próximo PC. Se a instrução for um desvio e soubermos qual deve ser o próximo PC, podemos ter uma penalidade de desvio de zero. Uma cache de previsão de desvio que armazena o endereço previsto para a próxima instrução após um desvio é chamada de *buffer de destino de desvio* ou *cache de destino de desvio*. A Figura 3.21 mostra um buffer de destino de desvio.

Como o buffer de destino de desvio prevê o endereço da próxima instrução e o envia *antes* de decodificar a instrução, *precisamos* saber se a instrução lida é prevista como um desvio tomado. Se o PC da instrução lida combinar com um PC no buffer de instrução, o PC previsto correspondente será usado como próximo PC. O hardware para esse buffer de destino de desvio é essencialmente idêntico ao hardware para a cache.

Se uma entrada correspondente for encontrada no buffer de destino de desvio, a busca começará imediatamente no PC previsto. Observe que, diferentemente de um buffer de previsão de desvio, a entrada da previsão precisa corresponder a essa instrução, pois o PC previsto será enviado antes de se saber sequer se essa instrução é um desvio. Se o

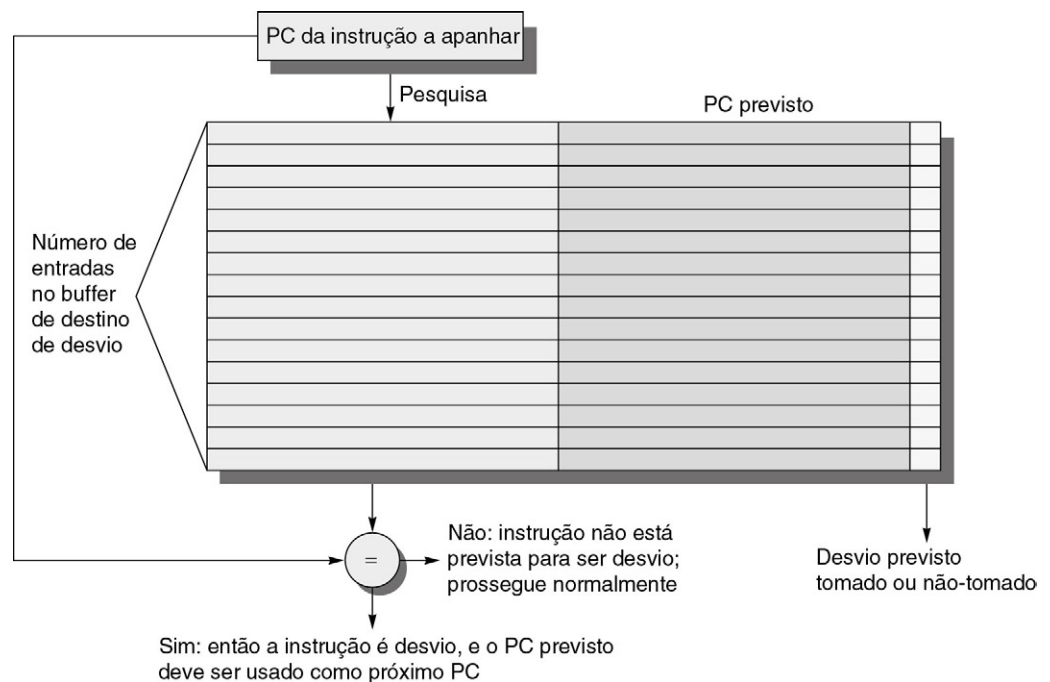



FIGURA 3.21 Um buffer de destino de desvio.
O PC da instrução sendo lida é combinado com um conjunto de endereços de instrução armazenados na primeira coluna; estes representam os endereços de desvios conhecidos. Se o PC for correspondente a uma dessas entradas, a instrução sendo lida será um desvio tomado, e o segundo campo, o PC previsto, conterá a previsão para o próximo PC após o desvio. A busca começa imediatamente nesse endereço. O terceiro campo, que é opcional, pode ser usado para os bits extras de status de previsão.

processador não verificasse se a entrada corresponde a esse PC, o PC errado seria enviado para instruções que não fossem desvios, resultando em um processador mais lento. Só precisamos armazenar os desvios tomados previstos no buffer de destino de desvio, pois um desvio não tomado deve simplesmente apanhar a próxima instrução sequencial como se não fosse um desvio.

A Figura 3.22 mostra as etapas detalhadas quando se usa um buffer de destino de desvio para um pipeline simples de cinco estágios. A partir disso, podemos ver que não haverá atraso de desvio se uma entrada de previsão de desvio for encontrada no buffer e a previsão estiver correta. Caso contrário, haverá uma penalidade de pelo menos dois ciclos de clock. Lidar com erros de previsão e faltas é um desafio significativo, pois normalmente teremos de interromper a busca da instrução enquanto reescrevemos a entrada do buffer. Assim, gostaríamos de tornar esse processo rápido para minimizar a penalidade.

Para avaliar se um buffer de destino de desvio funciona bem, primeiro temos de determinar as penalidades em todos os casos possíveis. A 3 contém essa informação para o pipeline simples de cinco estágios.

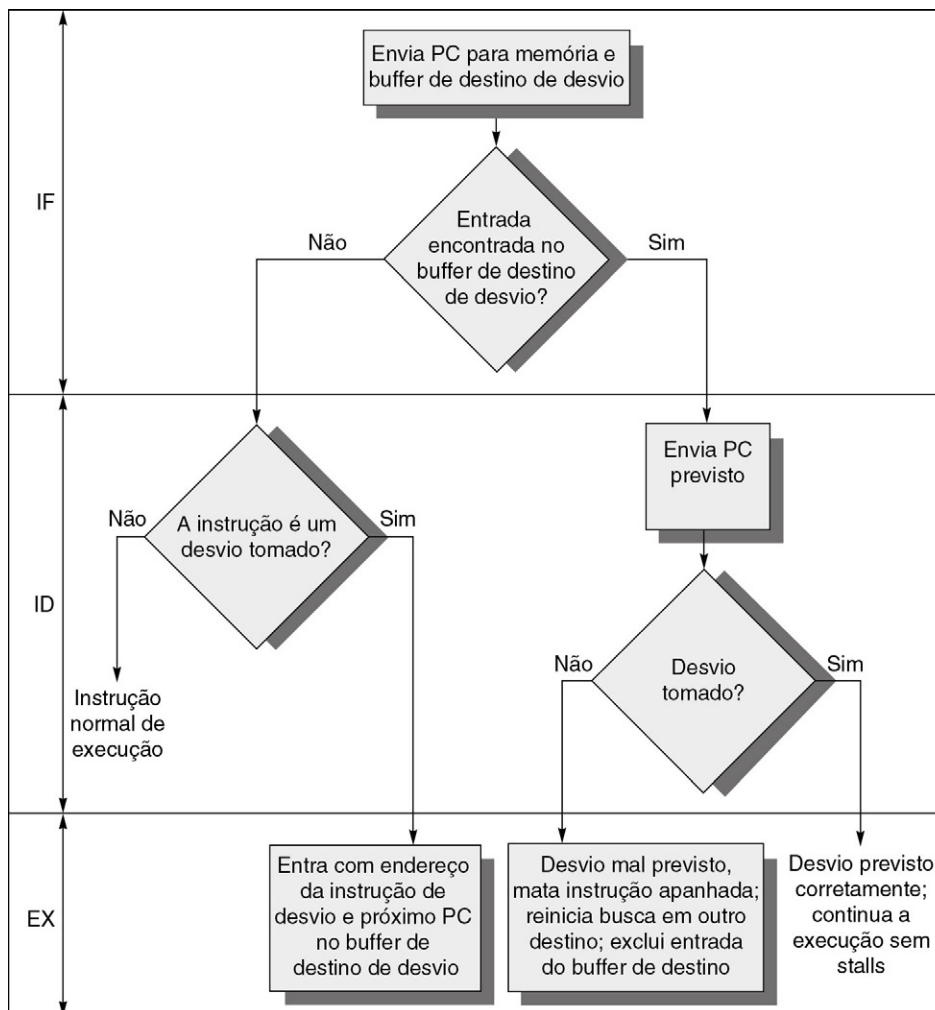



FIGURA 3.22 Etapas envolvidas no tratamento de uma instrução com um buffer de destino de desvio.

Exemplo Determine a penalidade de desvio total para um buffer de destino de desvio, considerando os ciclos de penalidade para os erros de previsão individuais da Figura 3.23. Faça as seguintes suposições sobre a precisão de previsão e taxa de acerto:
A precisão de previsão é de 90% (para instruções no buffer).
A taxa de acerto no buffer é de 90% (para desvios previstos tomados).

Resposta Calculamos a penalidade verificando a probabilidade de dois eventos: o desvio tem previsão de ser tomado, mas acaba não sendo tomado, e o desvio é tomado mas não é encontrado no buffer. Ambos carregam uma penalidade de dois ciclos.

$$\begin{aligned}\text{Probabilidade (desvio no buffer, mas não tomado realmente)} &= \text{Percentual de taxa de acerto do buffer}\\ &\quad \times \text{Porcentagem de previsões incorretas}\\ &= 90\% \times 10\% = 0{,}09\end{aligned}$$

$$\begin{aligned}\text{Probabilidade (desvio não no buffer, mas tomado)} &= 10\%\\ \text{Penalidade de desvio} &= (0{,}09 + 0{,}10) \times 2\\ \text{Penalidade de desvio} &= 0{,}38\end{aligned}$$

Essa penalidade se compara com uma penalidade de desvio para os desvios adiados, que avaliaremos no Apêndice C, de cerca de meio ciclo de clock por desvio. Lembre-se, no entanto, de que a melhoria da previsão de desvio dinâmico crescerá à medida que crescer o tamanho do pipeline, portanto, o atraso do desvio; além disso, previsores melhores gerarão uma vantagem de desempenho maior. Os processadores modernos de alto desempenho apresentam atrasos de previsão incorreta de desvio da ordem de 15 ciclos de clock. Obviamente, uma previsão precisa é essencial!

Uma variação no buffer de destino de desvio é armazenar uma ou mais *instruções-alvo* no lugar do *endereço-alvo* previsto ou adiconalmente a ele. Essa variação possui duas vantagens em potencial: 1) ela permite que o acesso ao buffer de desvio leve mais tempo do que o tempo entre buscas sucessivas de instruções, possivelmente permitindo um buffer de destino de desvio maior; 2) o armazenamento das instruções-alvo reais em buffer permite que realizemos uma otimização chamada *branch folding*. O *branch folding* pode ser usado para obter desvios incondicionais de 0 ciclo, e às vezes os desvios condicionais de 0 ciclo.

Considere um buffer de destino de desvio que coloca instruções no buffer a partir do caminho previsto e está sendo acessado com o endereço de um desvio incondicional. A única função do desvio incondicional é alterar o PC. Assim, quando o buffer de destino de desvio sinaliza um acerto e indica que o desvio é incondicional, o pipeline pode simplesmente substituir a instrução do buffer de destino de desvio no lugar da instrução que

Instrução no buffer	Previsão	Desvio real	Ciclos de Penalidade
Sim	Tomado	Tomado	0
Sim	Tomado	Não tomado	2
Não		Tomado	2
Não		Não tomado	0

FIGURA 3.23 Penalidades para todas as combinações possíveis de que o desvio está no buffer e o que ele realmente faz supondo que armazenemos apenas os desvios tomados no buffer.
Não existe penalidade de desvio se tudo for previsto corretamente e o desvio for encontrado no buffer de desvio. Se o desvio não for previsto corretamente, a penalidade será igual a um ciclo de clock para atualizar o buffer com a informação correta (durante o que uma instrução não poderá ser apanhada) e um ciclo de clock, se for preciso, para reiniciar a busca da próxima instrução correta para o desvio. Se o desvio não for encontrado e tomado, ocorrerá uma penalidade de dois ciclos enquanto o buffer for atualizado.

é retornada da cache (que é o desvio incondicional). Se o processador estiver enviando múltiplas instruções por ciclo, o buffer precisará fornecer múltiplas instruções para obter o máximo de benefício. Em alguns casos, talvez seja possível eliminar o custo de um desvio.

Previsões de endereço de retorno

Ao tentarmos aumentar a oportunidade e a precisão da especulação, encararemos o desafio de prever saltos indiretos, ou seja, saltos cujo endereço de destino varia durante a execução. Embora os programas em linguagem de alto nível gerem esses saltos para chamadas de procedimento indiretas, instruções select, case e os gotos do FORTRAN, a grande maioria dos saltos indiretos vem de retornos de procedimento. Por exemplo, para os benchmarks SPEC95, esses retornos são responsáveis por mais de 15% dos desvios e pela grande maioria dos saltos indiretos na média. Para linguagens orientadas a objeto, como C + + e Java, os retornos de procedimento são ainda mais frequentes. Assim, focar nos retornos de procedimento parece apropriado.

Embora os retornos de procedimento possam ser previstos com um buffer de destino de desvio, a exatidão dessa técnica de previsão pode ser baixa se o procedimento for chamado de vários locais e as chamadas de um local não forem agrupadas no tempo. Por exemplo, no SPEC CPU95, um previsor de desvio agressivo consegue uma precisão de menos de 60% para tais desvios de retorno. Para contornar esse problema, alguns projetos usam um pequeno buffer de endereços de retorno operando como uma pilha. Essa estrutura coloca em cache os endereços de retorno mais recentes: colocando um endereço de retorno na pilha em uma chamada e retirando-o em um retorno. Se a cache for suficientemente grande (ou seja, do mesmo tamanho da profundidade máxima de chamada), vai prever os retornos perfeitamente. A Figura 3.24

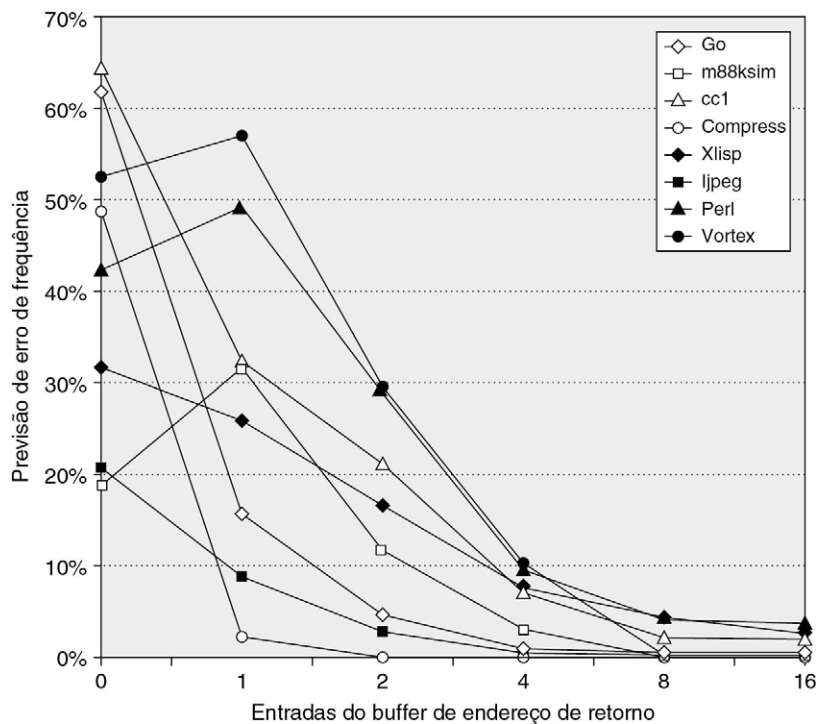



FIGURA 3.24 Exatidão da previsão para um buffer de endereço de retorno operado como pilha em uma série de benchmarks SPEC CPU95.

A precisão é a fração dos endereços de retorno previstos corretamente. Um buffer de 0 entrada implica que a previsão-padrão de desvio seja utilizada. Como as profundidades de chamadas normalmente são muito grandes, com algumas exceções, um buffer modesto funciona bem. Esse dado vem de Skadron *et al*. (1999) e utiliza um mecanismo de reparo para impedir a adulteração dos endereços de retorno em cache.

mostra o desempenho de um buffer de retorno desse tipo com 0-16 elementos para uma série de benchmarks SPEC CPU95. Usaremos um previsor de retorno semelhante quando examinarmos os estudos do ILP na Seção 3.10. Tanto os processadores Intel Core quanto os processadores AMD Phenom têm previsores de endereço de retorno.

Unidades integradas de busca de instrução

Para atender as demandas dos processadores de múltiplo despacho, muitos projetistas recentes escolheram implementar uma unidade integrada de busca de instrução, como uma unidade autônoma separada, que alimenta instruções para o restante do pipeline. Basicamente, isso significa reconhecer que não é mais válido caracterizar a busca de instruções como um único estágio da pipe, dadas as complexidades do múltiplo despacho.

Em vez disso, os projetos recentes usaram uma unidade integrada de busca de instrução que abrange diversas funções:

1. *Previsão integrada de desvio.* O previsor de desvio torna-se parte da unidade de busca de instrução e está constantemente prevendo desvios, de modo a controlar a busca no pipeline.
2. *Pré-busca de instrução.* Para oferecer múltiplas instruções por clock, a unidade de busca de instrução provavelmente precisará fazer a busca antecipadamente. A unidade controla autonomamente a pré-busca de instruções (ver uma discussão das técnicas para fazer isso no Capítulo 2), integrando com a previsão de desvio.
3. *Acesso à memória de instruções e armazenamento em buffer.* Ao carregar múltiplas instruções por ciclo, diversas complexidades são encontradas, incluindo a dificuldade de que a busca de múltiplas instruções pode exigir o acesso a múltiplas linhas de cache. A unidade de busca de instrução contorna essa complexidade, usando a pré-busca para tentar esconder o custo de atravessar blocos de cache. A unidade de busca de instrução também oferece o uso de buffer, basicamente atuando como uma unidade sob demanda, para oferecer instruções ao estágio de despacho conforme a necessidade e na quantidade necessária.

Hoje, quase todos os processadores sofisticados usam uma unidade de busca de instrução separada conectada ao resto do pipeline por um buffer contendo instruções pendentes.

Especulação: problemas de implementação e extensões

Nesta seção, exploraremos três ideias que envolvem a implementação da especulação, começando com o uso da renomeação de registradores, a técnica que substituiu quase totalmente o uso de um buffer de reordenação. Depois, discutiremos uma extensão possível importante para a especificação no fluxo de controle: uma ideia chamada *previsão de valor*.

Suporte à especulação: renomeação de registrador versus *buffers de reordenação*

Uma alternativa ao uso de um buffer de reordenação (ROB) é o uso explícito de um conjunto físico maior de registradores, combinado com a renomeação de registradores. Essa técnica se baseia no conceito de renomeação usado no algoritmo de Tomasulo e o estende. No algoritmo de Tomasulo, os valores dos *registradores arquitetonicamente visíveis a arquitetura* (R0, ..., R31 e F0, ..., F31) estão contidos, em qualquer ponto na execução, em alguma combinação do conjunto de registradores e a estações de reserva. Com o acréscimo da especulação, os valores de registrador também podem residir temporariamente no ROB. De qualquer forma, se o processador não enviar novas instruções por um período de tempo, todas as instruções existentes serão confirmadas, e os valores dos registradores

aparecerão no banco de registradores, que corresponde diretamente aos registradores visíveis arquitetonicamente.

Na técnica de renomeação de registrador, um conjunto estendido de registradores físicos é usado para manter os registradores arquitetonicamente visíveis e também valores temporários. Assim, os registradores estendidos substituem a função do ROB e das estações de reserva. Durante o despacho de instrução, um processo de renomeação mapeia os nomes dos registradores da arquitetura para os números dos registradores físicos no conjunto de registradores estendido, alocando um novo registrador não usado para o destino. Hazards WAW e WAR são evitados renomeando-se o registrador de destino, e a recuperação da especulação é tratada, porque um registrador físico, mantido como um destino de instrução não se torna o registrador da arquitetura até que a instrução seja confirmada. O mapa de renomeação é uma estrutura de dados simples que fornece o número de registrador físico do registrador correspondente ao registrador da arquitetura especificado. Essa estrutura é semelhante em estrutura e função à tabela de *status* de registrador no algoritmo de Tomasulo. Quando uma instrução é confirmada, a tabela restante é atualizada permanentemente para indicar que um registrador físico corresponde ao registrador de arquitetura real, finalizando efetivamente a atualização ao status do processador. Embora um ROB não seja necessário com a renomeação de registrador, o hardware deve rastrear instruções em uma estrutura similar à de uma fila e atualizar a tabela de renomeação em ordem estrita.

Uma vantagem da técnica de renomeação *versus* a técnica ROB é que a confirmação de instrução é simplificada, pois exige apenas duas ações simples: 1) registrar que o mapeamento entre um número de registrador da arquitetura e o número do registrador físico não é mais especulativo e 2) liberar quaisquer registradores físicos sendo usados para manter o valor "mais antigo" do registrador da arquitetura. Em um projeto com estações de reserva, uma estação é liberada quando a instrução que a utiliza termina a execução, e uma entrada ROB é liberada quando a instrução correspondente é confirmada.

Com a renomeação de registrador, a desalocação de registradores é mais complexa, pois, antes de liberarmos um registrador físico, temos de saber se ele não corresponde mais a um registrador da arquitetura e se nenhum outro uso do registrador físico está pendente. Um registrador físico corresponde a um registrador da arquitetura até que o registrador da arquitetura seja reescrito, fazendo com que a tabela de renomeação aponte para outro lugar. Ou seja, se nenhuma entrada restante apontar para determinado registrador físico, ela não corresponderá mais a um registrador da arquitetura. Porém, ainda poderá haver usos pendentes do registrador físico. O processador poderá determinar se esse é o caso examinando os especificadores de registrador de origem de todas as instruções nas filas da unidade funcional. Se determinado registrador físico não aparecer como origem e não for designado como registrador da arquitetura, ele poderá ser reclamado e realocado.

Como alternativa, o processador pode simplesmente esperar até que seja confirmada outra instrução que escreva no mesmo registrador da arquitetura. Nesse ponto, pode não haver mais usos para o valor pendente antigo. Embora esse método possa amarrar um registrador físico por um pouco mais de tempo do que o necessário, ele é fácil de implementar e, portanto, é usado em vários superescalares recentes.

Uma pergunta que você pode estar fazendo é: "Como sabemos quais registradores são de arquitetura se eles estão constantemente mudando?" Na maior parte do tempo em que um programa está executando, isso não importa. Porém, existem casos em que outro processo, como o sistema operacional, precisa ser capaz de saber exatamente onde reside o conteúdo de certo registrador de arquitetura. Para entender como essa capacidade é fornecida, considere que o processador não envia instruções por algum período de tempo. Por fim,

todas as instruções no pipeline serão confirmadas, e o mapeamento entre os registradores arquitetonicamente visíveis e os registradores físicos se tornará estável. Nesse ponto, um subconjunto dos registradores físicos contém os registradores arquitetonicamente visíveis, e o valor de qualquer registrador físico não associado a um registrador de arquitetura é desnecessário. Então, é fácil mover os registradores de arquitetura para um subconjunto fixo de registradores físicos, de modo que os valores possam ser comunicados a outro processo.

Tanto a renomeação de registrador quanto os buffers de reorganização continuam a ser usados em processadores sofisticados, que hoje podem ter em ação até 40-50 instruções (incluindo loads e stores aguardando na cache). Seja usando a renomeação, seja usando um buffer de reorganização, o principal gargalo para a complexidade de um superescalar escalonado dinamicamente continua sendo o despacho de conjuntos de instruções com dependências dentro do conjunto. Em particular, instruções dependentes em um conjunto de despacho devem ser enviadas com os registradores virtuais designados das instruções das quais eles dependem. Uma estratégia para despacho de instrução com renomeação de registrador similar ao usado para múltiplo despacho com buffers de reorganização (página 157) pode ser empregada como a seguir:

1. A lógica de despacho pré-reserva registradores físicos suficientes para todo o conjunto de despachos (digamos, quatro registradores para um conjunto de quatro instruções com, no máximo, um resultado de registrador por instrução).
2. A lógica de despacho determina quais dependências existem dentro do conjunto. Se não existir nenhuma dependência dentro do conjunto, a estrutura de renomeação de registrador será usada para determinar o registrador físico que contém, ou vai conter, o resultado do qual a instrução depende. Quando não existe nenhuma dependência dentro do conjunto, o resultado é um conjunto de despachos anterior, e a tabela de renomeação de registrador terá o número de registrador correto.
3. Se uma instrução depender de uma instrução anterior no conjunto, o registrador físico pré-reservado no qual o resultado será colocado será usado para atualizar a informação para a instrução que está enviando.

Observe que, assim como no caso do buffer de reorganização, a lógica de despacho deve determinar as dependências dentro do conjunto e atualizar as tabelas de renomeação em um único clock e, como antes, a complexidade de fazer isso para grande número de instruções por clock torna-se uma grande limitação na largura do despacho.

Quanto especular

Uma das vantagens significativas da especulação é a sua capacidade de desvendar eventos que, de outra forma, fariam com que o pipeline ficasse em stall mais cedo, como as falhas de cache. Porém, essa vantagem em potencial possui uma significativa desvantagem em potencial. A especulação não é gratuita: ela gasta tempo e energia, e a recuperação da especulação incorreta reduz ainda mais o desempenho. Além disso, para dar suporte à taxa mais alta de execução de instrução, necessária para se tirar proveito da especulação, o processador precisa ter recursos adicionais, que exigem área de silício e energia. Finalmente, se a especulação levar a um evento excepcional, como a falhas de cache ou TLB, o potencial para uma perda significativa de desempenho aumentará se esse evento não tiver ocorrido sem especulação.

Para manter a maior parte da vantagem enquanto minimiza as desvantagens, a maioria dos pipelines com especulação só permite que eventos excepcionais de baixo custo (como uma falha de cache de primeiro nível) sejam tratados no modo especulativo. Se houver um evento excepcional dispendioso, como falha de cache de segundo nível ou falha do buffer de TLB, o processador vai esperar até que a instrução que causa o evento deixe de ser

especulativa, antes de tratar dele. Embora isso possa degradar ligeiramente o desempenho de alguns programas, evita perdas de desempenho significativa em outros, especialmente naqueles que sofrem com a alta frequência de tais eventos, acoplado com uma previsão de desvio menos que excelente.

Na década de 1990, as desvantagens em potencial da especulação eram menos óbvias. Com a evolução dos processadores, os custos reais da especulação se tornaram mais aparentes, e as limitações do despacho mais amplo e a especulação se tornaram óbvias. Retomaremos essa questão em breve.

Especulação por desvios múltiplos

Nos exemplos que consideramos neste capítulo, tem sido possível resolver um desvio antes de ter de especular outro. Três situações diferentes podem beneficiar-se com a especulação em desvios múltiplos simultaneamente: 1) frequência de desvio muito alta; 2) agrupamento significativo de desvios; e 3) longos atrasos nas unidades funcionais. Nos dois primeiros casos, conseguir um desempenho alto pode significar que múltiplos desvios são especulados e até mesmo tratar de mais de um desvio por clock. Os programas de banco de dados, e outras computações com inteiros menos estruturadas, geralmente exigem essas propriedades, tornando a especulação em desvios múltiplos mais importante. De modo semelhante, longos atrasos nas unidades funcionais podem aumentar a importância da especulação em desvios múltiplos como um meio de evitar stalls a partir de atrasos de pipeline mais longos.

A especulação em desvios múltiplos complica um pouco o processo de recuperação da especulação, mas é simples em outros aspectos. Em 2011, nenhum processador havia combinado especulação completa com a resolução de múltiplos desvios por ciclo, e é improvável que os custos de fazer isso fossem justificados em termos de desempenho *versus* complexidade e energia.

Especulação e o desafio da eficiência energética

Qual é o impacto da especulação sobre a eficiência energética? À primeira vista, pode-se argumentar que usar a especulação sempre diminui a eficiência energética, já que sempre que a especulação está errada ela consome energia em excesso de dois modos:

1. As instruções que foram especuladas e aquelas cujos resultados não foram necessários geraram excesso de trabalho para o processador, desperdiçando energia.
2. Desfazer a especulação e restaurar o status do processador para continuar a execução no endereço apropriado consome energia adicional que não seria necessária sem especulação.

Certamente, a especulação vai aumentar o consumo de energia e, se pudermos controlar a especulação, será possível medir o custo (ou pelo menos o custo da potência dinâmica). Mas, se a especulação diminuir o tempo de execução mais do que aumentar o consumo médio de energia, a energia total consumida pode ser menor.

Assim, para entender o impacto da especulação sobre a eficiência energética, precisamos examinar com que frequência a especulação leva a um trabalho desnecessário. Se número significativo de instruções desnecessárias for executado, é improvável que a especulação melhore em comparação com o tempo de execução! A Figura 3.25 mostra a fração de instruções executadas a partir da especulação incorreta. Como podemos ver, essa fração é pequena em códigos científicos e significativa (cerca de 30% em média) em códigos inteiros. Assim, é improvável que a especulação seja eficiente em termos de energia para aplicações de números inteiros. Os projetistas devem evitar a especulação, tentar reduzir

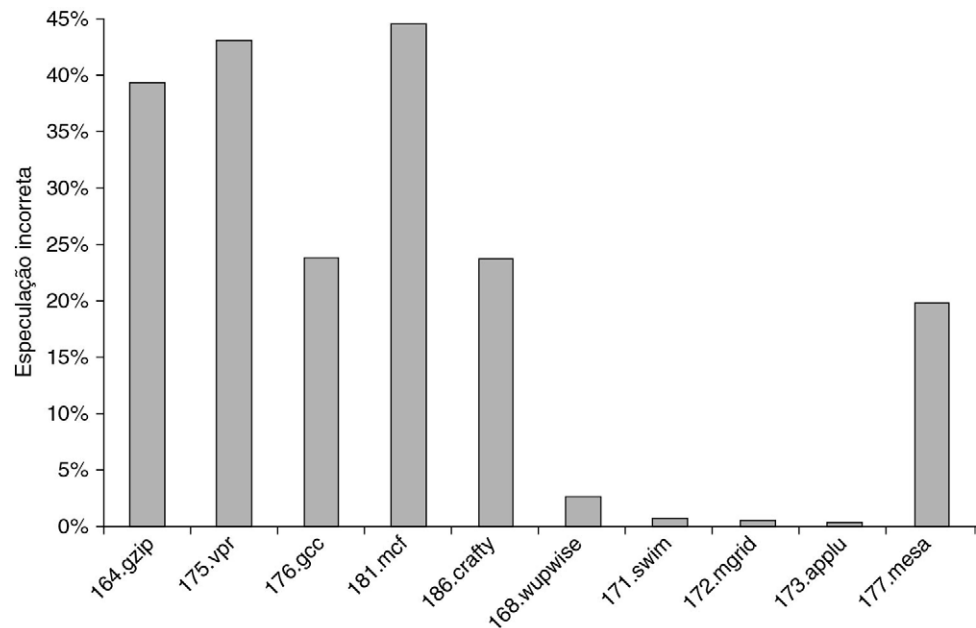



FIGURA 3.25 Fração de instruções que são executadas como resultado de especulação incorreta para programas inteiros (os cinco primeiros) em comparação a programas de PF (os cinco últimos).

a especulação incorreta ou pensar em novas abordagens, como somente especular em desvios altamente previsíveis.

Previsão de valor

Uma técnica para aumentar a quantidade de ILP disponível em um programa é a previsão de valor. A *previsão de valor* tenta prever o valor que será produzido por uma instrução. Obviamente, como a maioria das instruções produz um valor diferente toda vez que é executada (ou, pelo menos, um valor diferente a partir de um conjunto de valores), a previsão de valor só pode ter sucesso limitado. Portanto, existem certas instruções para as quais é mais fácil prever o valor resultante — por exemplo, loads que carregam de um pool constante ou que carregam um valor que muda com pouca frequência. Além disso, quando uma instrução produz um valor escolhido a partir de um pequeno conjunto de valores em potencial, é possível prever o valor resultante correlacionando-o com outros comportamentos do programa.

A previsão de valor é útil quando aumenta significativamente a quantidade de ILP disponível. Isso é mais provável quando um valor é usado como origem de uma cadeia de computações dependentes, como em um load. Como a previsão de valor é usada para melhorar especulações e a especulação incorreta tem impacto prejudicial no desempenho, a exatidão da previsão é essencial.

Embora muitos pesquisadores tenham se concentrado na previsão de valor nos últimos 10 anos, os resultados nunca foram atraentes o suficiente para justificar sua incorporação em processadores reais. Em vez disso, uma ideia simples e mais antiga, relacionada com a previsão de valor, tem sido adotada: a previsão de aliasing de endereço. *Previsão de aliasing de endereço* é uma técnica simples que prevê se dois stores ou um load e um store se referem ao mesmo endereço de memória. Se duas referências desse tipo não se referirem ao mesmo endereço, elas poderão ser seguramente trocadas. Caso contrário, teremos de esperar até que sejam conhecidos os endereços de memória acessados pelas instruções. Como não precisamos realmente prever os valores de endereço somente se tais valores entram em

conflito, a previsão é mais estável e mais simples. Essa forma limitada de especulação de valor de endereço tem sido usada em diversos processadores e pode tornar-se universal no futuro.

3.10 ESTUDOS DAS LIMITAÇÕES DO ILP

A exploração do ILP para aumentar o desempenho começou com os primeiros processadores com pipeline na década de 1960. Nos anos 1980 e 1990, essas técnicas foram fundamentais para conseguir rápidas melhorias de desempenho. A questão de quanto ILP existe foi decisiva para a nossa capacidade de aumentar, a longo prazo, o desempenho a uma taxa que ultrapassasse o aumento na velocidade da tecnologia básica do circuito integrado. Em uma escala mais curta, a questão crítica do que é necessário para explorar mais ILP é crucial para os projetistas de computadores e de compiladores. Os dados apresentados nesta seção também nos oferecem um modo de examinar o valor das ideias que introduzimos no capítulo anterior, incluindo a falta de ambiguidade de memória, renomeação de registrador e especulação.

Nesta seção, vamos rever um dos estudos feitos sobre essas questões (baseados no estudo de Wall, em 1993). Todos esses estudos do paralelismo disponível operam fazendo um conjunto de suposições e vendo quanto paralelismo está disponível sob essas suposições. Os dados que examinamos aqui são de um estudo que faz o mínimo de suposições; na verdade, é provável que o modelo de hardware definitivo não seja realizável. Apesar disso, todos esses estudos consideram certo nível de tecnologia de compilador, e algumas dessas suposições poderiam afetar os resultados, apesar do uso de um hardware incrivelmente ambicioso.

Como veremos, para modelos de hardware que tenham custo razoável, é improvável que os custos de uma especulação muito agressiva possam ser justificados. As ineficiências energéticas e o uso de silício são simplesmente muito altos. Enquanto muitos na comunidade de pesquisa e os principais fabricantes de processadores estavam apostando em maior exploração do ILP, e foram inicialmente relutantes em aceitar essa possibilidade, em 2005 eles foram forçados a mudar de ideia.

O modelo de hardware

Para ver quais poderiam ser os limites do ILP, primeiro precisamos definir um processador ideal. Um processador ideal é aquele do qual são removidas todas as restrições sobre o ILP. Os únicos limites sobre o ILP em tal processador são aqueles impostos pelos fluxos de dados reais, pelos registradores ou pela memória.

As suposições feitas para um processador ideal ou perfeito são as seguintes:

1. *Renomeação infinita de registrador.* Existe um número infinito de registradores virtuais à disposição e, por isso, todos os hazards WAW e WAR são evitados e um número ilimitado de instruções pode iniciar a execução simultaneamente.
2. *Previsão perfeita de desvio.* A previsão de desvio é perfeita. Todos os desvios condicionais são previstos com exatidão.
3. *Previsão perfeita de salto.* Todos os saltos (incluindo o salto por registrador, usado para retorno e saltos calculados) são perfeitamente previstos. Quando combinado com a previsão de desvio perfeita, isso é equivalente a ter um processador com especulação perfeita e um buffer ilimitado de instruções disponíveis para execução.
4. *Análise perfeita de alias de endereço de memória.* Todos os endereços de memória são conhecidos exatamente, e um load pode ser movido antes de um store, desde que os endereços não sejam idênticos. Observe que isso implementa a análise de alias de endereço perfeita.

5. *Caches perfeitas*. Todos os endereços de memória utilizam um ciclo de clock. Na prática, os processadores superescalares normalmente consumirão grande quantidade de ILP ocultando falhas de cache, tornando esses resultados altamente otimistas.

As suposições 2 e 3 eliminam *todas* as dependências de controle. De modo semelhante, as suposições 1 e 4 eliminam *todas menos as dependências de dados verdadeiras*. Juntas, essas quatro suposições significam que qualquer instrução na execução do programa pode ser escalonada no ciclo imediatamente após a execução da predecessora da qual depende. É possível ainda, sob essas suposições, que a *última* instrução executada dinamicamente no programa seja sempre escalonada no primeiro ciclo de clock! Assim, esse conjunto de suposições substitui a especulação de controle e endereço e as implementa como se fossem perfeitas.

Inicialmente, examinamos um processador que pode enviar um número ilimitado de instruções ao mesmo tempo, olhando arbitrariamente para o futuro da computação. Para todos os modelos de processador que examinamos, não existem restrições sobre quais tipos de instruções podem ser executadas em um ciclo. Para o caso de despacho ilimitado, isso significa que pode haver um número ilimitado de loads ou stores enviados em um ciclo de clock. Além disso, todas as latências de unidade funcional são consideradas como tendo um ciclo, de modo que qualquer sequência de instruções dependentes pode ser enviada em ciclos sucessivos. As latências maiores do que um ciclo diminuiriam o número de despachos por ciclo, embora não o número de instruções em execução em qualquer ponto (as instruções em execução em qualquer ponto normalmente são chamadas de *instruções in flight*).

É evidente que esse processador está às margens do irrealizável. Por exemplo, o IBM Power7 (ver Wendell *et al*., 2010) é o processador superescalar mais avançado anunciado até o momento. O Power7 envia até seis instruções por clock e inicia a execução em até 8-12 unidades de execução (somente duas das quais são unidades de load/store), suporta um grande conjunto de registradores renomeados (permitindo centenas de instruções no ato), usa um previsor de desvio grande e agressivo, e emprega a desambiguação de memória dinâmica. O Power7 continuou o movimento rumo ao uso de mais paralelismo no nível de thread, aumentando a largura do suporte para multithreading simultâneo (SMT) para quatro threads por núcleo e o número de núcleos por chip para oito. Depois de examinar o paralelismo disponível para o processador perfeito, examinaremos o que pode ser alcançado em qualquer processador que seja projetado em futuro próximo.

Para medir o paralelismo disponível, um conjunto de programas foi compilado e otimizado com os compiladores de otimização MIPS padrão. Os programas foram instrumentados e executados para produzir um rastro das referências de instruções e dados. Cada uma dessas instruções é então escalonada o mais cedo possível, limitada apenas pelas dependências de dados. Como um rastro é usado, a previsão de desvio perfeita e a análise de alias perfeita são fáceis de fazer. Com esses mecanismos, as instruções podem ser escalonadas muito mais cedo do que poderiam ser de outra forma, movendo grande quantidade de instruções, cujos dados não são dependentes, incluindo desvios, que são perfeitamente previstos.

A Figura 3.26 mostra a quantidade média de paralelismo disponível para seis dos benchmarks SPEC92. Por toda esta seção, o paralelismo é medido pela taxa de despacho de instrução média. Lembre-se de que todas as instruções possuem uma latência de um ciclo; uma latência maior reduziria o número médio de instruções por clock. Três desses

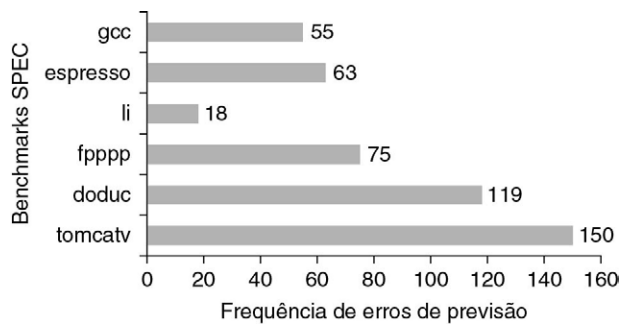



FIGURA 3.26 ILP disponível em um processador perfeito para seis dos benchmarks SPEC92.
Os três primeiros programas são programas de inteiros, o os últimos três são programas de ponto flutuante. Os programas de ponto flutuante são intensos em termos de loop e têm grande quantidade de paralelismo em nível de loop.

benchmarks (fpppp, doduc e tomcatv) utilizam intensamente números de ponto flutuante, e os outros três são programas para inteiros. Dois dos benchmarks de ponto flutuante (fpppp e tomcatv) possuem paralelismo extensivo, que poderia ser explorado por um computador vetorial ou por um multiprocessador (porém, a estrutura do fpppp é muito confusa, pois algumas transformações foram feitas manualmente no código). O programa doduc possui extenso paralelismo, mas esse paralelismo não ocorre em loops paralelos simples, assim como no fpppp e tomcatv. O programa li é um interpretador LISP que possui dependências muito curtas.

Limitações do ILP para processadores realizáveis

Nesta seção examinaremos o desempenho de processadores com níveis ambiciosos de suporte ao hardware igual ou melhor que o disponível em 2011 ou dados os eventos e lições da última década, que provavelmente estarão disponíveis num futuro próximo. Em particular, supomos os seguintes atributos fixos:

1. Até 64 despachos de instrução por clock *sem* restrições de despacho ou cerca de 10 vezes a amplitude de despacho total do maior processador em 2011. Conforme examinaremos mais adiante, as implicações práticas de grandes amplitudes de despacho na frequência do clock, complexidade lógica e potência podem ser a limitação mais importante da exploração do ILP.
2. Um previsor de torneio com 1 K entradas e um previsor de retorno de 16 entradas. Esse previsor é bastante comparável aos melhores previsores em 2011; o previsor não é um gargalo importante.
3. A desambiguidade perfeita de referências de memória feita dinamicamente é bastante ambiciosa, mas talvez seja possível para pequenos tamanhos de janela (e, portanto, pequenas taxas de despacho e buffers de load-store) ou através de um previsor de dependência de memória.
4. Renomeação de registradorres com 64 registradores adicionais de inteiros e 64 de ponto flutuante, o que é ligeiramente menor do que o processador mais agressivo em 2011. O Intel Core i7 tem 128 entradas em seu buffer de reordenação, embora eles não sejam divididos entre inteiros e ponto flutuante, enquanto o IBM Power7 tem quase 200. Observe que supomos uma latência de pipeline de um ciclo que reduz significativamente a necessidade de entradas de buffer de reorganização. Tanto o Power7 quanto o i7 têm latências de 10 ciclos ou mais.

A Figura 3.27 mostra o resultado para essa configuração à medida que variamos o tamanho da janela. Essa configuração é mais complexa e dispendiosa do que quaisquer implementações existentes, sobretudo em termos do número de despachos de instrução, que é mais de 10 vezes maior que o maior número de despachos disponíveis em qualquer processador em 2011. Apesar disso, oferece um limite útil sobre o que as implementações futuras poderiam alcançar. Os dados nessa figura provavelmente são muito otimistas por outro motivo. Não existem restrições de despacho entre as 64 instruções: todas elas podem ser referências de memória. Ninguém sequer contemplaria essa capacidade em um processador no futuro próximo. Infelizmente, é muito difícil vincular o desempenho de um processador com restrições de despacho razoáveis; não só o espaço de possibilidades é muito grande, mas a existência de restrições de despacho exige que o paralelismo seja avaliado com um escalonador de instrução preciso, o que torna muito dispendioso o custo de estudo de processadores com grande quantidade de despachos.

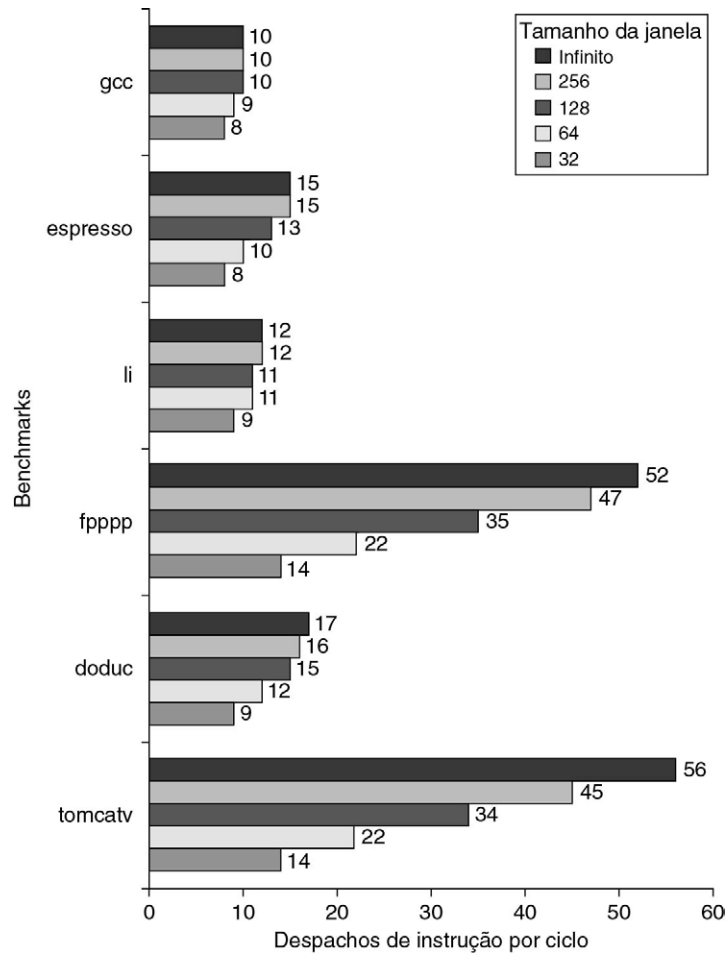



FIGURA 3.27 A quantidade de paralelismo disponível em comparação ao tamanho da janela para diversos programas de inteiros e ponto flutuante com até 64 despachos arbitrários de instruções por clock.
Embora haja menos registradores restantes do que o tamanho da janela, o fato de que todas as operações tenham latência de um ciclo e o número de registradores restantes seja igual à largura de despacho permite ao processador explorar o paralelismo dentro de toda a janela. Na implementação real, o tamanho da janela e o número de registradores restantes devem ser equilibrados para impedir que um desses fatores restrinja demais a taxa de despacho.

Além disso, lembre-se de que, na interpretação desses resultados, as falhas de cache e as latências não unitárias não foram levadas em consideração, e esses dois efeitos terão impacto significativo!

A observação mais surpreendente da Figura 3.27 é que, mesmo com as restrições de processador realista listadas, o efeito do tamanho da janela para os programas de inteiros não é tão severo quanto para programas de ponto flutuante. Esse resultado aponta para a principal diferença entre esses dois tipos de programas. A disponibilidade do paralelismo em nível de loop em dois dos programas de ponto flutuante significa que a quantidade de ILP que pode ser explorado é mais alta, mas que, para programas de inteiros, outros fatores — como a previsão de desvio, a renomeação de registrador e menos paralelismo, para começar — são limitações importantes. Essa observação é crítica, devido à ênfase aumentada no desempenho para inteiros nos últimos anos. Na realidade, a maior parte do crescimento do mercado na última década — processamento de transação, servidores Web e itens semelhantes — dependeu do desempenho para inteiros, em vez do ponto flutuante. Conforme veremos na seção seguinte, para um processador realista em 2011, os níveis de desempenho reais são muito inferiores àqueles mostrados na Figura 3.27.

Dada a dificuldade de aumentar as taxas de instrução com projetos de hardware realistas, os projetistas enfrentam um desafio na decisão de como usar melhor os recursos limitados disponíveis em um circuito integrado. Uma das escolhas mais interessantes é entre processadores mais simples com caches maiores e taxas de clock mais altas *versus* mais ênfase no paralelismo em nível de instrução com clock mais lento e caches menores. O exemplo a seguir ilustra os desafios, e no Capítulo 4 veremos uma técnica alternativa para explorar o paralelismo fino na forma de GPUs.

Exemplo Considere os três processadores hipotéticos (mas não atípicos) a seguir, nos quais vamos executar o benchmark gcc do SPEC:

1. Um pipe estático MIPS de duplo despacho rodando a uma frequência de clock de 4 GHz e alcançando um CPI de pipeline de 0,8. Esse processador tem um sistema de cache que gera 0,005 falha por instrução.
2. Uma versão fortemente canalizada de um processador MIPS com duplo despacho com cache ligeiramente menor e frequência de clock de 5 GHz. O CPI de pipeline do processador é 1,0, e as caches menores geram 0,0055 falha por instrução, em média.
3. Um superescalar especulativo com uma janela de 64 entradas. Ele alcança metade da taxa de despacho ideal medida para esse tamanho de janela (usar dados da Figura 3.27). Esse processador tem as menores caches, que levam a 0,01 falha por instrução, mas oculta 25% da penalidade de falha a cada falha através de escalonamento dinâmico. Esse processador tem um clock de 2,5 GHz.

Suponha que o tempo da memória principal (que estabelece a penalidade de falha) seja de 50 ns. Determine o desempenho relativo desses três processadores.

Resposta Primeiro, usaremos as informações de penalidade de falha e taxa de falha para calcular a contribuição para o CPI a partir das falhas de cache para cada configuração. Faremos isso com a seguinte fórmula:

$$\text{CPI de cache} = \text{Falhas por instrução} \times \text{Penalidade de falha}$$

Precisamos calcular as penalidades de falha para cada sistema:

$$\text{Penalidade de falha} = \frac{\text{Tempo de acesso à memória}}{\text{Ciclo de clock}}$$

Os tempos de ciclo de clock são de 250 ps, 200 ps e 400 ps, respectivamente. Portanto, as penalidades de falha são

$$\text{Penalidade de falha}_1 = \frac{50\text{ ns}}{250\text{ ps}} = 200\text{ ciclos}$$

$$\text{Penalidade de falha}_2 = \frac{50\text{ ns}}{200\text{ ps}} = 250\text{ ciclos}$$

$$\text{Penalidade de falha}_3 = \frac{0{,}75 \times 50\text{ ns}}{400\text{ ps}} = 94\text{ ciclos}$$

Aplicando isso a cada cache:

$$\text{CPI de cache}_1 = 0{,}005 \times 200 = 1{,}0$$

$$\text{CPI de cache}_2 = 0{,}0055 \times 250 = 1{,}4$$

$$\text{CPI de cache}_3 = 0{,}01 \times 94 = 0{,}94$$

Conhecemos a contribuição do CPI de pipeline para tudo, exceto para o processador 3. Esse CPI de pipeline é dado por:

$$\text{CPI de pipeline}_3 = \frac{1}{\text{Taxa de despacho}} = \frac{1}{9 \times 0{,}5} = \frac{1}{4{,}5} = 0{,}22$$

Agora podemos encontrar o CPI para cada processador adicionando as contribuições de CPI do pipeline e cache.

$$\text{CPI}_1 = 0{,}8 + 1{,}0 = 1{,}8$$

$$\text{CPI}_2 = 1{,}0 + 1{,}4 = 2{,}4$$

$$\text{CPI}_3 = 0{,}22 + 0{,}94 = 1{,}16$$

Uma vez que essa é a mesma arquitetura, podemos comparar as taxas de execução de instrução em milhões de instruções por segundo (MIPS) para determinar o desempenho relativo:

$$\text{Taxa de execução de instrução} = \frac{\text{CR}}{\text{CPI}}$$

$$\text{Taxa de execução de instrução}_1 = \frac{4.000\text{ MHz}}{1{,}8} = 2.222\text{ MIPS}$$

$$\text{Taxa de execução de instrução}_2 = \frac{5.000\text{ MHz}}{2{,}4} = 2.083\text{ MIPS}$$

$$\text{Taxa de execução de instrução}_3 = \frac{2.500\text{ MHz}}{1{,}16} = 2.155\text{ MIPS}$$

Neste exemplo, o superescalar estático simples de duplo despacho parece ser o melhor. Na prática, o desempenho depende das suposições de CPI e frequência do clock.

Além dos limites deste estudo

Como em qualquer estudo de limite, o estudo que examinaremos nesta seção tem suas próprias limitações. Nós as dividimos em duas classes: 1) limitações que surgem até mesmo para o processador especulativo perfeito e 2) limitações que surgem de um ou mais modelos realistas. Naturalmente, todas as limitações na primeira classe se aplicam à segunda. As limitações mais importantes que se aplicam até mesmo ao modelo perfeito são:

1. *Hazards WAW e WAR através da memória.* O estudo eliminou hazards WAW e WAR por meio da renomeação de registrador, mas não no uso da memória. Embora a princípio tais circunstâncias possam parecer raras (especialmente os hazards WAW), elas surgem devido à alocação de frames de pilha. Uma chamada de procedimento

reutiliza os locais da memória de um procedimento anterior na pilha, e isso pode levar a hazards WAW e WAR, que são desnecessariamente limitadores. Austin e Sohi (1992) examinam essa questão.

2. *Dependências desnecessárias.* Com um número infinito de registradores, todas as dependências, exceto as verdadeiras dependências de dados de registrador, são removidas. Porém, algumas dependências surgem de recorrências ou de convenções de geração de código que introduzem dependências de dados verdadeiras desnecessárias. Um exemplo disso é a dependência da variável de controle em um simples loop *for*. Como a variável de controle é incrementada a cada iteração do loop, ele contém pelo menos uma dependência. Como mostraremos no Apêndice H, o desdobramento de loop e a otimização algébrica agressiva podem remover essa computação dependente. O estudo de Wall inclui quantidade limitada dessas otimizações, mas sua aplicação de forma mais agressiva poderia levar a maior quantidade de ILP. Além disso, certas convenções de geração de código introduzem dependências desnecessárias, em particular o uso de registradores de endereço de retorno e de um registrador para o ponteiro de pilha (que é incrementado e decrementado na sequência de chamada/retorno). Wall remove o efeito do registrador de endereço de retorno, mas o uso de um ponteiro de pilha na convenção de ligação pode causar dependências "desnecessárias". Postiff *et al.* (1999) exploraram as vantagens de remover essa restrição.
3. *Contornando o limite de fluxo de dados.* Se a previsão de valor funcionou com alta precisão, ela poderia contornar o limite do fluxo de dados. Até agora, nenhum dos mais de 100 trabalhos sobre o assunto conseguiu melhoria significativa no ILP usando um esquema de previsão realista. Obviamente, a previsão perfeita do valor de dados levaria ao paralelismo efetivamente infinito, pois cada valor de cada instrução poderia ser previsto *a priori*.

Para um processador menos que perfeito, várias ideias têm sido propostas e poderiam expor mais ILP. Um exemplo é especular ao longo de vários caminhos. Essa ideia foi tratada por Lam e Wilson (1992) e explorada no estudo abordado nesta seção. Especulando sobre caminhos múltiplos, o custo da recuperação incorreta é reduzido e mais paralelismo pode ser desvendado. Só faz sentido avaliar esse esquema para um número limitado de desvios, pois os recursos de hardware exigidos crescem exponencialmente. Wall (1993) oferece dados para especular nas duas direções em até oito desvios. Dados os custos de perseguir os dois caminhos mesmo sabendo que um deles será abandonado (e a quantidade crescente de computação inútil, na medida em que tal processo é seguido por desvios múltiplos), cada projeto comercial, em vez disso, dedicou um hardware adicional para melhor especulação sobre o caminho correto.

É fundamental entender que nenhum dos limites mencionados nesta seção é fundamental no sentido de que contorná-los exige uma mudança nas leis da Física! Em vez disso, eles são limitações práticas que implicam a existência de algumas barreiras formidáveis para a exploração do ILP adicional. Essas limitações — sejam elas tamanho de janela, sejam detecção de alias ou previsão de desvio — representam desafios para projetistas e pesquisadores contornarem.

Tentativas de romper esses limites nos primeiros cinco anos deste século resultaram em frustrações. Algumas técnicas levaram a pequenas melhorias, porém muitas vezes com significativos aumentos em complexidade, aumentos no ciclo de clock e aumentos desproporcionais na potência. Em resumo, os projetistas descobriram que tentar extrair mais ILP era simplesmente ineficiente. Vamos retomar essa discussão nos "Comentários Finais".

3.11 QUESTÕES CRUZADAS: TÉCNICAS DE ILP E O SISTEMA DE MEMÓRIA

Especulação de hardware *versus* especulação de software

As técnicas de uso intenso de hardware para a especulação, mencionadas no Capítulo 2, e as técnicas de software do Apêndice H oferecem enfoques alternativos à exploração do ILP. Algumas das escolhas e suas limitações para esses enfoques aparecem listadas a seguir:

- Para especular extensivamente, temos de ser capazes de tirar a ambiguidade das referências à memória. Essa capacidade é difícil de fazer em tempo de compilação para programas de inteiros que contêm ponteiros. Em um esquema baseado no hardware, a eliminação da ambiguidade dos endereços de memória em tempo de execução dinâmica é feita com o uso das técnicas que vimos para o algoritmo de Tomasulo. Essa desambiguidade nos permite mover loads para depois de stores em tempo de execução. O suporte para referências de memória especulativas pode contornar o conservadorismo do compilador, mas, a menos que essas técnicas sejam usadas cuidadosamente, o overhead dos mecanismos de recuperação poderá sobrepor as vantagens.
- A especulação baseada em hardware funciona melhor quando o fluxo de controle é imprevisível e quando a previsão de desvio baseada em hardware é superior à previsão de desvio baseada em software, feita em tempo de compilação. Essas propriedades se mantêm para muitos programas de inteiros. Por exemplo, um bom previsor estático tem taxa de erro de previsão de cerca de 16% para quatro principais programas SPEC92 de inteiros, e um previsor de hardware tem taxa de erro de previsão de menos de 10%. Como as instruções especuladas podem atrasar a computação quando a previsão é incorreta, essa diferença é significativa. Um resultado dessa diferença é que mesmo os processadores escalonados estaticamente normalmente incluem previsores de desvio dinâmicos.
- A especulação baseada em hardware mantém um modelo de exceção completamente preciso, até mesmo para instruções especuladas. As técnicas recentes baseadas em software têm acrescentado suporte especial para permitir isso também.
- A especulação baseada em hardware não exige código de compensação ou de manutenção, que é necessário para mecanismos ambiciosos de especulação de software.
- Técnicas baseadas em compilador podem se beneficiar com a capacidade de ver adiante na sequência de código, o que gera melhor escalonamento de código do que uma técnica puramente controlada pelo hardware.
- A especulação baseada em hardware com escalonamento dinâmico não exige sequências de código diferentes para conseguir bom desempenho para diferentes implementações de uma arquitetura. Embora essa vantagem seja a mais difícil de quantificar, ela pode ser a mais importante com o passar do tempo. Interessante é que essa foi uma das motivações para o IBM 360/91. Por outro lado, arquiteturas explicitamente paralelas mais recentes, como IA-64, acrescentaram uma flexibilidade que reduz a dependência de hardware inerente em uma sequência de código.

As principais desvantagens do suporte à especulação no hardware são a complexidade e os recursos de hardware adicionais exigidos. Esse custo de hardware precisa ser avaliado contra a complexidade de um compilador, para uma técnica baseada em software, e a quantidade e a utilidade das simplificações, em um processo que conte com tal compilador.

Alguns projetistas tentaram combinar as técnicas dinâmica e baseada em compilador para conseguir o melhor de cada uma. Essa combinação pode gerar interações interessantes e

obscuras. Por exemplo, se moves condicionais forem combinados com a renomeação de registrador, aparecerá um efeito colateral sutil. Um move condicional que é anulado ainda precisa copiar um valor para o registrador de destino, pois foi renomeado no pipeline de instruções. Essas interações sutis complicam o processo de projeto e verificação, e também podem reduzir o desempenho.

O processador Intel Itanium foi o computador mais ambicioso já projetado com base no suporte de software ao ILP e à especulação. Ele não concretizou as esperanças dos projetistas, especialmente para códigos de uso geral e não científicos. Conforme as ambições dos projetistas para explorar o ILP foram reduzidas à luz das dificuldades discutidas na Seção 3.10, a maioria das arquiteturas se estabeleceu em mecanismos baseados em hardware com taxas de despacho de três ou quatro instruções por clock.

Execução especulativa e o sistema de memória

Inerente a processadores que suportam execução especulativa ou instruções condicionais é a possibilidade de gerar endereços inválidos que não existiriam sem execução especulativa. Não só isso seria um comportamento incorreto se fossem tomadas exceções de proteção, mas os benefícios da execução especulativa também seriam sobrepujados pelo overhead de exceções falsas. Portanto, o sistema de memória deve identificar instruções executadas especulativamente e instruções executadas condicionalmente e suprimir a exceção correspondente.

Seguindo um raciocínio similar, não podemos permitir que tais instruções façam com que a cache sofra stall em uma falha, porque stalls desnecessários poderiam sobrepujar os benefícios da especulação. Portanto, esses processadores devem ser associados a caches sem bloqueio.

Na verdade, a penalidade em uma falha em L2 é tão grande que, normalmente, os compiladores só especulam sobre falhas em L1. A Figura 2.5, na página 72, mostra que para alguns programas científicos bem comportados o compilador pode sustentar múltiplas falhas pendentes em L2 a fim de cortar efetivamente a penalidade de falha de L2. Novamente, para que isso funcione, o sistema de memória por trás da cache deve fazer a correspondência entre os objetivos do compilador em número de acessos simultâneos à memória.

3.12 MULTITHREADING: USANDO SUPORTE DO ILP PARA EXPLORAR O PARALELISMO EM NÍVEL DE THREAD

O tópico que cobrimos nesta seção, o multithreading, é na verdade um tópico cruzado, uma vez que tem relevância para o pipelining e para os superescalares, para unidades de processamento gráfico (Cap. 4) e para multiprocessadores (Cap. 5). Apresentaremos o tópico aqui e exploraremos o uso do multithreading para aumentar o throughput de uniprocessador usando múltiplos threads para ocultar as latências de pipeline e memória. No Capítulo 4, vamos ver como o multithreading fornece as mesmas vantagens nas GPUs e, por fim, o Capítulo 5 vai explorar a combinação de multithreading e multiprocessamento. Esses tópicos estão firmemente interligados, já que o multithreading é uma técnica primária usada para expor mais paralelismo para o hardware. Num senso estrito, o multithreading usa paralelismo em nível de thread e, por isso, é o assunto do Capítulo 5, mas seu papel tanto na melhoria da utilização de pipelines quanto nas GPUs nos motiva a introduzir aqui esse conceito.

Embora aumentar o desempenho com o uso do ILP tenha a grande vantagem de ser razoavelmente transparente para o programador, como já vimos, o ILP pode ser bastante

limitado ou difícil de explorar em algumas aplicações. Em particular, com taxas razoáveis de despacho de instrução, as falhas de cache que vão para a memória ou caches fora do chip provavelmente não serão ocultadas por um ILP disponível. Obviamente, quando o processador está paralisado esperando por uma falha de cache, a utilização das unidades funcionais cai drasticamente.

Uma vez que tentativas de cobrir paralisações longas de memória com mais ILP tem eficácia limitada, é natural perguntar se outras formas de paralelismo em uma aplicação poderiam ser usadas para ocultar atrasos de memória. Por exemplo, um sistema de processamento on-line de transações tem paralelismo natural entre as múltiplas pesquisas e atualizações que são apresentadas pelas requisições. Sem dúvida, muitas aplicações científicas contêm paralelismo natural, uma vez que muitas vezes modelam a estrutura tridimensional, paralela, da natureza, e essa estrutura pode ser explorada usando threads separados. Mesmo aplicações de desktop que usam sistemas operacionais modernos baseados no Windows muitas vezes têm múltiplas aplicações ativas sendo executadas, proporcionando uma fonte de paralelismo.

O *multithreading* permite que vários threads compartilhem as unidades funcionais de um único processador em um padrão superposto. Em contraste, um método mais geral de explorar o *paralelismo em nível de thread* (TLP) é com um multiprocessador que tenha múltiplos threads independentes operando ao mesmo tempo e em paralelo. O multithreading, entretanto, não duplica todo o processador, como ocorre em um multiprocessador. Em vez disso, o multithreading compartilha a maior parte do núcleo do processador entre um conjunto de threads, duplicando somente o status privado, como os registradores e o contador de programa. Como veremos no Capítulo 5, muitos processadores recentes incorporam múltiplos núcleos de processador em um único chip e também fornecem multithreading dentro de cada núcleo.

Duplicar o status por thread de um núcleo de processador significa criar um banco de registradores separado, um PC separado e uma tabela de páginas separada para cada thread. A própria memória pode ser compartilhada através dos mecanismos de memória virtual, que já suportam multiprogramação. Além disso, o hardware deve suportar a capacidade de mudar para um thread diferente com relativa rapidez. Em particular, uma mudança de thread deve ser mais eficiente do que uma mudança de processador, que em geral requer de centenas a milhares de ciclos de processador. Obviamente, para o hardware em multithreading atingir melhoras de desempenho, um programa deve conter múltiplos threads (às vezes dizemos que a aplicação é *multithreaded*) que possam ser executados de modo simultâneo. Esses threads são identificados por um compilador (em geral, a partir de uma linguagem com construções para paralelismo) ou pelo programador.

Existem três técnicas principais para o multithreading. O *multithreading de granularidade fina* alterna os threads a cada clock, fazendo com que a execução de múltiplos threads seja intercalada. Essa intercalação normalmente é feita em um padrão *round-robin*, pulando quaisquer threads que estejam em stall nesse momento. Para tornar o multithreading fino prático, a CPU precisa ser capaz de trocar de threads a cada ciclo de clock. A principal vantagem do multithreading de granularidade fina é que ele pode esconder as falhas de throughput que surgem de stalls curtos e longos, pois as instruções de outros threads podem ser executadas quando um thread está em stall. A principal desvantagem do multithreading de granularidade fina é que ele atrasa a execução dos threads individuais, pois um thread que estiver pronto para ser executado sem stalls será atrasado pelas instruções de outros threads. Ela troca um aumento no throughput do multithreading por uma perda no desempenho (como medido pela latência) de um único thread. O processador Sun

Niagara, que vamos examinar a seguir, usa multithreading de granularidade fina, assim como as GPUs Nvidia, que examinaremos no Capítulo 4.

O *multithreading* de granularidade *grossa* foi inventado como alternativa para o multithreading de granularidade fina. O multithreading de granularidade grossa troca de threads somente em stalls dispendiosos, como as falhas de cache de nível 2. Essa troca alivia a necessidade da comutação de threads ser essencialmente livre e é muito menos provável que atrase o processador, pois as instruções de outros threads só serão enviadas quando um thread encontrar um stall dispendioso.

Contudo, o multithreading de granularidade grossa apresenta uma grande desvantagem: sua capacidade de contornar as falhas de throughput, especialmente de stalls mais curtos, é limitada. Essa limitação advém dos custos de partida do pipeline de multithreading de granularidade grossa. Como uma CPU com multithreading de granularidade grossa envia instruções de um único thread quando ocorre um stall, o pipeline precisa ser esvaziado ou congelado. O novo thread que inicia a execução após o stall precisa preencher o pipeline antes que as instruções sejam capazes de terminar. Devido a esse overhead de partida, o multithread de granularidade grossa é muito mais útil para reduzir as penalidades dos stalls de alto custo, quando o preenchimento do pipeline é insignificante em comparação com o tempo do stall. Muitos projetos de pesquisa têm explorado o multithreading de granularidade grossa, mas nenhum grande processador atual usa essa técnica.

A implementação mais comum do multithreading é chamada *multithreading simultâneo* (SMT). O multithreading simultâneo é uma variação do multithreading de granularidade *grossa* que surge naturalmente quando é implementado em um processador de múltiplo despacho, escalonado dinamicamente. Assim como ocorre com outras formas de multithreading, o SMT usa o paralelismo em nível de thread para ocultar eventos com grande latência em um processador, aumentando o uso das unidades funcionais. A principal característica do SMT é que a renomeação de registrador e o escalonamento dinâmico permitem que múltiplas instruções de threads independentes sejam executadas sem considerar as dependências entre eles; a resolução das dependências pode ser manipulada pela capacidade de escalonamento dinâmico.

A Figura 3.28 ilustra conceitualmente as diferenças na capacidade de um processador explorar os recursos de um superescalar para as seguintes configurações de processador:

- Um superescalar sem suporte para multithreading
- Um superescalar com multithreading de granularidade grossa
- Um superescalar com multithreading de granularidade fina
- Um superescalar com multithreading simultâneo

No superescalar sem suporte para multithreading, o uso dos slots de despacho é limitado pela falha de ILP, incluindo ILP para ocultar a memória de latência. Devido ao comprimento das falhas das caches L2 e L3, grande parte do processador pode ficar ociosa.

No superescalar com multithreading de granularidade grossa, os stalls longos são parcialmente escondidos pela troca por outro thread que usa os recursos do processador. Embora isso reduza o número de ciclos de clock completamente ociosos, dentro de cada ciclo de clock as limitações do ILP ainda levam a ciclos ociosos. Além do mais, como em um processador com multithreading de granularidade grossa a troca de thread só ocorre quando existe um stall e o novo thread possui um período de partida, provavelmente restarão alguns ciclos totalmente ociosos.

No caso de granularidade fina, a intercalação de threads elimina slots totalmente vazios. Além disso, já que o thread que envia instruções é mudado a cada ciclo de clock, operações

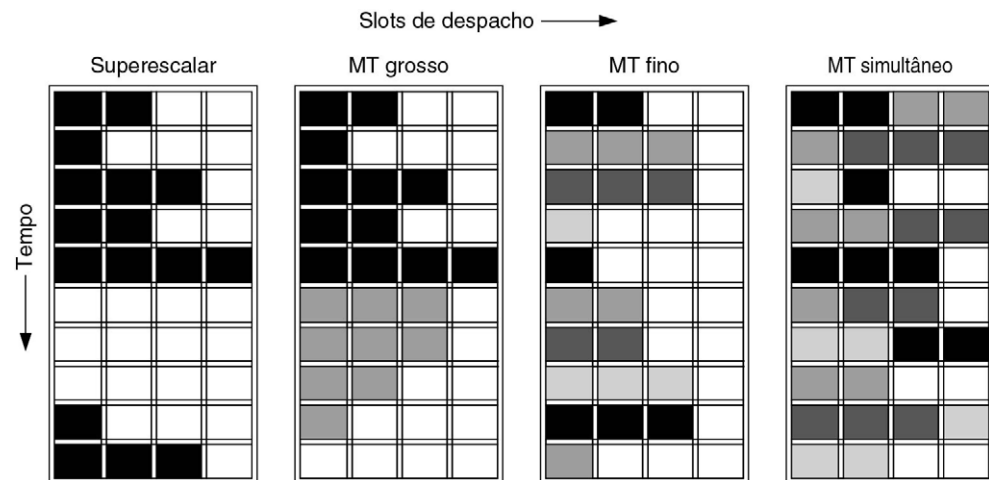



FIGURA 3.28 **Como quatro técnicas diferentes utilizam os slots de despacho de um processador superescalar.**

A dimensão horizontal representa a capacidade de despacho de instrução em cada ciclo de clock. A dimensão vertical representa uma sequência de ciclos de clock. Uma caixa vazia (branca) indica que o slot de despacho correspondente não é usado nesse ciclo de clock. Os tons de cinza e preto correspondem a quatro threads diferentes nos processadores de multithreading. Preto também é usado para indicar os slots de despacho ocupados no caso do superescalar sem suporte para multithreading. Sun T1 e T2 (também conhecidos como Niagara) são processadores multithread de granularidade fina, enquanto os processadores Intel Core i7 e IBM Power7 usam SMT. O T2 possuiu oito threads, o Power7 tem quatro, e o Intel i7 tem dois. Em todos os SMTs existentes, as instruções são enviadas de um thread por vez. A diferença do SMT é que a decisão subsequente de executar uma instrução é desacoplada e pode executar as operações vindo de diversas instruções diferentes no mesmo ciclo de clock.

com maior latência podem ser ocultadas. Como o despacho de instrução e a execução estão conectados, um thread só pode enviar as instruções que estiverem prontas. Em uma pequena largura de despacho isso não é um problema (um ciclo está ocupado ou não), porque o multithreading de granularidade fina funciona perfeitamente para um processador de despacho único e o SMT não faria sentido. Na verdade, no Sun T2 existem duplos despachos por clock, mas eles são de threads diferentes. Isso elimina a necessidade de implementar a complexa técnica de escalonamento dinâmico e, em vez disso, depende de ocultar a latência com mais threads.

Se um threading de granularidade fina for implementado sobre um processador com escalonamento dinâmico, de múltiplos despachos, o resultado será SMT. Em todas as implementações SMT existentes, todos os despachos vêm de um thread, embora instruções de threads diferentes possam iniciar sua execução no mesmo ciclo, usando o hardware de escalonamento dinâmico para determinar que instruções estão prontas. Embora a Figura 3.28 simplifique bastante a operação real desses processadores, ela ilustra as vantagens de desempenho em potencial do multithreading em geral e do SMT em particular, em processadores escalonáveis dinamicamente.

Multithreading simultâneos usam a característica de que um processador escalonado dinamicamente já tem muitos dos mecanismos de hardware necessários para dar suporte ao mecanismo, incluindo um grande conjunto de registradores virtual. O multithreading pode ser construído sobre um processador fora de ordem, adicionando uma tabela de renomeação por thread, mantendo PCs separados e fornecendo a capacidade de confirmar instruções de múltiplos threads.

Eficácia do multithreading de granularidade fina no Sun T1

Nesta seção, usaremos o processador Sun T1 para examinar a capacidade do multithreading de ocultar a latência. O T1 é um multiprocessador multicore com multithreading de granularidade fina introduzido pela Sun em 2005. O que torna o T1 especialmente interessante é que ele é quase totalmente focado na exploração do paralelismo em nível de thread (TLP) em vez do paralelismo em nível de instrução (ILP). O T1 abandonou o foco em ILP (pouco depois os processadores ILP mais agressivos serem introduzidos), retornou à estratégia simples de pipeline e focou na exploração do TLP, usando tanto múltiplos núcleos como multithreading para produzir throughput.

Cada processador T1 contém oito núcleos de processador, cada qual dando suporte a quatro threads. Cada núcleo de processador consiste em um pipeline simples de seis estágios e despacho único (um pipeline RISC padrão de cinco estágios, como aquele do Apêndice C, com um estágio a mais para a comutação de threads). O T1 utiliza o multithreading de granularidade fina (mas não SMT), passando para um novo thread a cada ciclo de clock, e os threads que estão ociosos por estarem esperando devido a um atraso no pipeline ou falha de cache são contornados no escalonamento. O processador fica ocioso somente quando os quatro threads estão ociosos ou em stall. Tanto loads quanto desvios geram um atraso de três ciclos, que só pode ser ocultado por outros threads. Um único conjunto de unidades funcionais de ponto flutuante é compartilhado pelos oito núcleos, pois o desempenho de ponto flutuante não foi o foco para o T1. A Figura 3.29 resume o processador T1.

Desempenho do multithreading do T1 de núcleo único

O T1 faz do TLP o seu foco, através do multithreading em um núcleo individual, e também através do uso de muitos núcleos simples em um único substrato. Nesta seção, vamos examinar a eficácia do T1 em aumentar o desempenho de um único núcleo através de multithreading de granularidade fina. No Capítulo 5, vamos voltar a examinar a eficácia de combinar multithreading com múltiplos núcleos.

Examinamos o desempenho do T1 usando três benchmarks orientados para servidor: TPC-C, SPECJBB (o SPEC Java Business Benchmark) e SPECWeb99. Já que múltiplos threads aumentam as demandas de memória de um único processador, eles poderiam sobrecarregar o sistema de memória, levando a reduções no ganho em potencial do multithreading. A Figura 3.30 mostra o aumento relativo na taxa de falha e a latência quando rodamos com um thread por núcleo em comparação a rodar quatro threads por núcleo para o TPC-C. As taxas de falha e as latências de falha aumentam, devido à maior

Característica	Sun T1
Multiprocessador e suporte para multithreading	Oito núcleos por chip; quatro threads por núcleo. Escalonamento de threads de granularidade fina. Uma unidade de ponto flutuante compartilhada para oito núcleos. Suporta apenas multiprocessamento no chip.
Estrutura de pipeline	Pipeline simples, em ordem, com profundidade de seis e atrasos de três ciclos para loads e desvios.
Caches L1	Instruções de 16 KB; 8 KB para dados. Tamanho de bloco de 64 bytes. Falhas em L2 usam 23 ciclos, sem considerar contenção.
Caches L2	Quatro caches L2 separadas, cada qual com 750 KB e associadas a um banco de memória. Tamanho de bloco de 64 bytes. A falha na memória principal usa 110 ciclos de clock, sem considerar contenção.
Implementação inicial	Processo de 90 nm; frequência do clock máxima de 1,2 GHz; potência 79 W; 300 M transistores, die de 379 mm^2.

FIGURA 3.29 Resumo do processador T1.

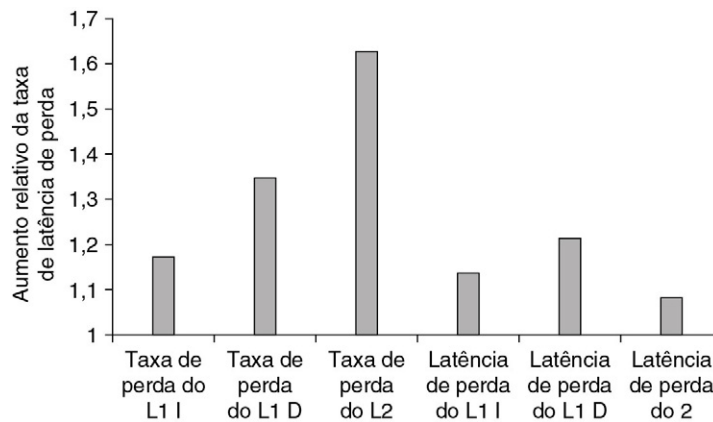



FIGURA 3.30 **Mudança relativa nas taxas de falha e latências de falha ao executar com um thread por núcleo contra quatro threads por núcleo no benchmark TPC-C.**
As latências são o tempo real para retornar os dados solicitados após uma falha. No caso de quatro threads, a execução dos outros threads poderia ocultar grande parte dessa latência.

contenção no sistema de memória. O aumento relativamente pequeno na latência de falha indica que o sistema de memória ainda tem capacidade não utilizada.

Ao examinarmos o comportamento de uma média de threads, podemos entender a interação entre multithreading e processamento paralelo. A Figura 3.31 mostra a porcentagem de ciclos para os quais um thread está executando, pronto mas não executando e não pronto. Lembre-se de que "não pronto" não significa que o núcleo com esse thread está em stall; somente quando todos os quatro threads estiverem não prontos é que o núcleo gerará um stall.

Os threads podem estar não prontos devido a falhas de cache, atrasos de pipeline (surgindo de instruções de longa latência, como desvios, loads, ponto flutuante ou multiplicação/divisão de inteiros) e uma série de efeitos menores. A Figura 3.32 mostra a frequência relativa dessas várias causas. Os efeitos de cache são responsáveis pelo thread não estando pronto em 50-75% do tempo, com falhas de instrução L1, falhas de dados L1 e falhas

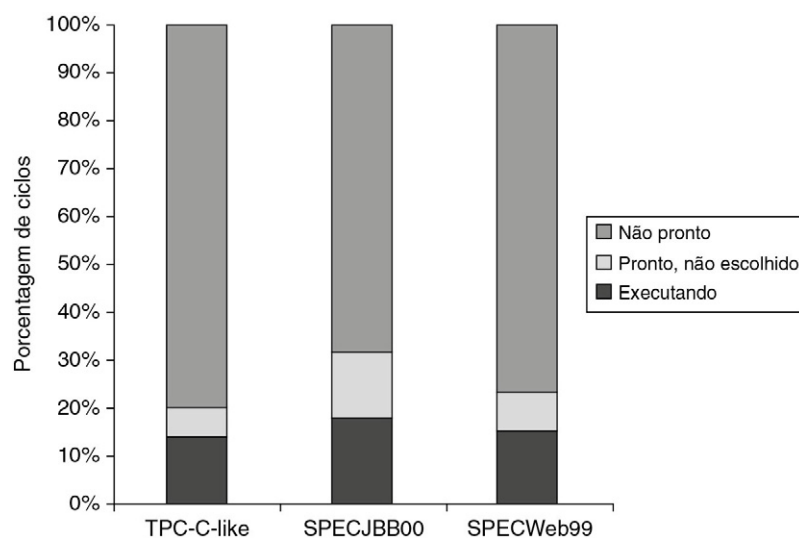



FIGURA 3.31 **Desmembramento de um thread mediano.**
"Executando" indica que o thread envia uma instrução nesse ciclo. "Pronto, não escolhido" significa que ele poderia ser enviado, mas outro thread foi escolhido, e "não pronto" indica que o thread está esperando o término de um evento (um atraso de pipeline ou falha de cache, por exemplo).

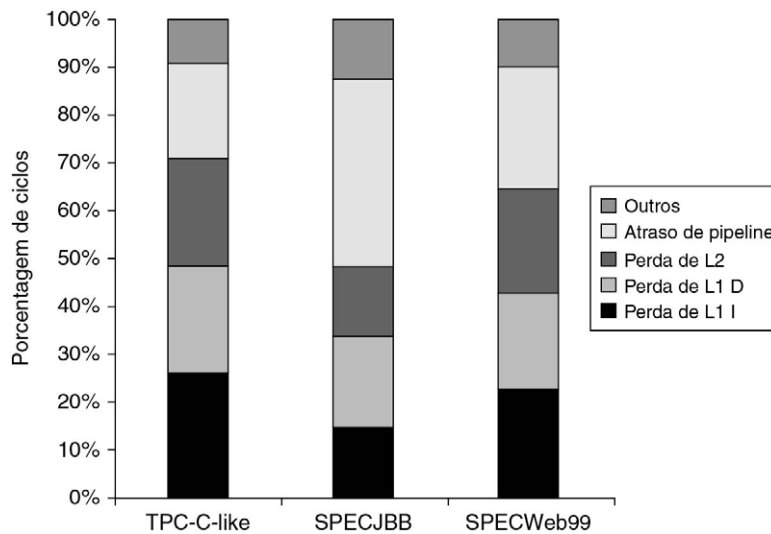



FIGURA 3.32 **Desmembramento de causas para um thread não pronto.**
A contribuição para a categoria "outros" varia. No TPC-C, o buffer de armazenamento cheio é o contribuinte maior; no SPEC-JBB, as instruções indivisíveis são o contribuinte maior; e, no SPECWeb99, ambos os fatores contribuem.

L2 contribuindo de forma aproximadamente igual. Os atrasos em potencial do pipeline (chamados "atrasos de pipeline") são mais severos no SPECJBB e podem surgir de sua frequência de desvio mais alta.

A Figura 3.33 mostra o CPI por thread e por núcleo. Já que o T1 é um processador multithreaded de granularidade fina com quatro threads por núcleo, com paralelismo suficiente, o CPI ideal por thread seria quatro, uma vez que isso significaria que cada thread estaria consumindo um a cada quatro ciclos. O CPI ideal por núcleo seria um. Em 2005, o IPC para esses benchmarks, sendo executados em núcleos ILP agressivos, seria similar ao visto em um núcleo T1. Entretanto, o núcleo T1 tinha tamanho muito modesto em comparação com os núcleos ILP agressivos de 2005, porque o T1 tinha oito núcleos em comparação aos dois a quatro oferecidos por outros processadores da mesma época. Como resultado, em 2005, quando foi lançado, o processador Sun T1 tinha o mesmo desempenho em aplicações de número inteiro com TLP extensivo e desempenho de memória exigente, como o SPECJBB e cargas de trabalho de processamento de transação.

Eficácia no multithreading simultâneo em processadores superescalares

Uma pergunta-chave é: "Quanto desempenho pode ser ganho com a implementação do SMT?" Quando essa pergunta foi explorada em 2000-2001, os pesquisadores presumiram que os superescalares dinâmicos ficariam maiores nos cinco anos seguintes, admitindo 6-8 despachos por clock com escalonamento dinâmico especulativo, muitos loads e stores

Benchmark	CPI por thread	CPI por núcleo
TPC-C	7,2	1,80
SPECJBB	5,6	1,40
SPECWeb99	6,6	1,65

FIGURA 3.33 **O CPI por thread, o CPI por núcleo, o CPI efetivo para oito núcleos e o IPC eficiente (inverso da CPI) para o processador T1 de oito núcleos.**

simultâneos, grandes caches primárias e 4-8 contextos com busca simultânea de múltiplos contextos. Mas nenhum processador chegou perto desse nível.

Em consequência, os resultados de pesquisa de simulação que mostraram ganhos para cargas de trabalho multiprogramadas de duas ou mais vezes são irrealistas. Na prática, as implementações existentes do SMT oferecem dois contextos com busca de apenas um, além de capacidades de despacho mais modestas. O resultado disso é que o ganho do SMT também é mais modesto.

Por exemplo, no Pentium 4 Extreme, implementado nos servidores HP-Compaq, o uso do SMT gera melhoria de desempenho de 1,01 quando executa o benchmark SPECintRate e cerca de 1,07 quando executa o benchmark SPECfpRate. Em um estudo separado, Tuck e Tullsen (2003) observam que os benchmarks paralelos SPLASH informam ganhos de velocidade de 1,02-1,67, com ganho médio de velocidade de cerca de 1,22.

Com a disponibilidade de medidas completas e esclarecedoras recém-feitas por Esmaeilzadeh *et al*. (2011), podemos examinar os benefícios de desempenho e energia de empregar SMT em um único núcleo i7 usando um conjunto de aplicações multithreaded. Os benchmarks que usamos consistem em uma coleção de aplicações científicas paralelas e um conjunto de programas Java multithreaded dos conjuntos DaCapo e SPEC Java, como resumido na Figura 3.34, que mostra a taxa de desempenho e a taxa de eficiência energética dos benchmarks executados em um núcleo do i7 com o SMT desligado e ligado.Figura 3.35

blackscholes	Determina os preços de um portfólio de opções com o PDE Black-Scholes
bodytrack	Rastreia um corpo humano sem marcas
canneal	Minimiza o custo de roteamento de um chip utilizando o algoritmo cache-awrae simulted annealing
facesim	Simula os movimentos de uma face humana para fins de visualização
ferret	Mecanismo de busca que encontra um conjunto de imagens similar a uma imagem pesquisada
fluidanimate	Simula a física do movimento dos fluidos para animação com um algoritmo SPH
raytrace	Usa simulação física para visualização
streamcluster	Computa uma aproximação para o agrupamento ótimo de pontos de dados
swaptions	Determina o preço de um portfólio de opções de troca com o framework Heath-Jarrow-Morton
vips	Aplica uma série de transformações em uma imagem
x264	Codificador de vídeo MPG-4 AVC/H.264
eclipse	Ambiente de desenvolvimento integrado
lusearch	Ferramenta de busca de texto
sunflow	Sistema de renderização fotorrealista
tomcat	Container de servlet Tomcat
tradebeans	Benchmark Tradebeans Daytrader
xalan	Um processador XSLT para transformar documentos XML
pjbb2005	Versão da SPEC JBB2005 (mas concentrada no tamanho do problema em vez do tempo)

FIGURA 3.34 Benchmarks paralelos usados aqui para examinar multithreading e também no Capítulo 5 para examinar o multiprocessamento com um i7.

A metade superior da figura consiste em benchmarks PARSEC coletados por Biena *et al.* (2008). Os benchmarks PARSEC foram criados para indicar aplicações paralelas intensas em termos de computação, que seriam apropriadas para processadores multicore. A metade inferior consiste em benchmarks Java multithreaded do conjunto DaCapo (ver Blackburn *et al.*, 2006) e pjbb2005 da SPEC. Todos esses benchmarks contêm algum paralelismo. Outros benchmarks Java nas cargas de trabalho DaCapo e SPEC Java usam threads múltiplos, mas têm pouco ou nenhum paralelismo e, portanto, não são usados aqui. Ver informações adicionais sobre as características desses benchmarks em relação às medidas aqui e no Capítulo 5 (Esmaeilzadeh *et al.*, 2011).

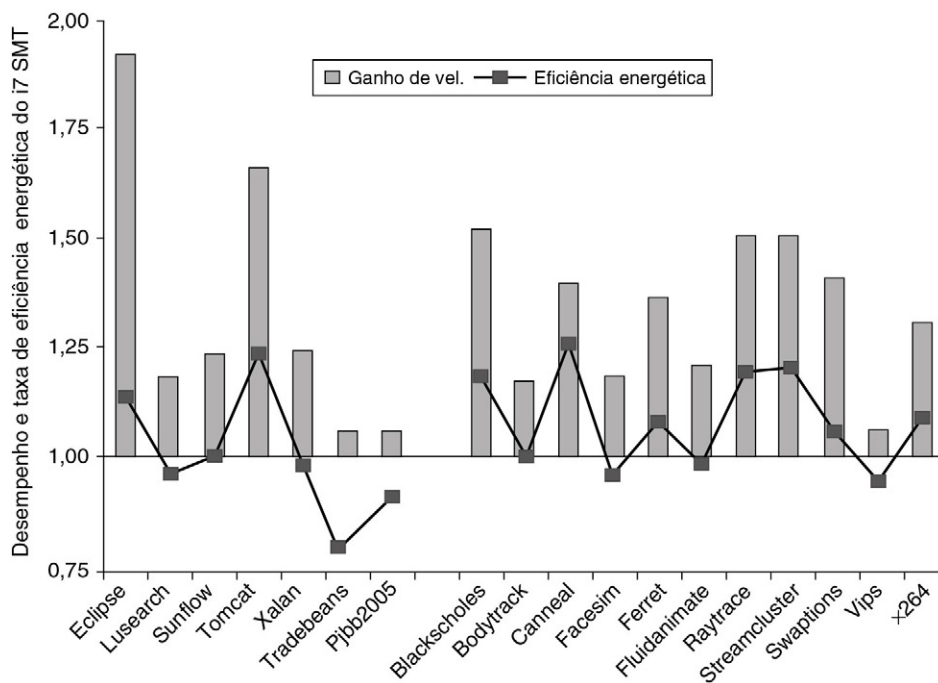



FIGURA 3.35 O ganho de velocidade usando multithreading em um núcleo de um processador i7 é, em média, de 1,28 para os benchmarks Java e de 1,31 para os benchmarks PARSEC (usando uma média harmônica não ponderada, que implica uma carga de trabalho em que o tempo total gasto executando cada benchmark no conjunto-base de thread único seja o mesmo).

A eficiência energética tem médias de 0,99 e 1,07, respectivamente (usando a média harmônica). Lembre-se de que qualquer coisa acima de 1,0 para a eficiência energética indica que o recurso reduz o tempo de execução mais do que aumenta a potência média. Dois dos benchmarks Java experimentam pouco ganho de velocidade e, por causa disso, têm efeito negativo sobre a eficiência energética. O Turbo Boost está desligado em todos os casos. Esses dados foram coletados e analisados por Esmaeilzadeh *et al.* (2011) usando o *build* Oracle (Sun) Hotspot 16.3-b01 Java 1.6.0 Virtual Machine e o compilador nativo gcc v4.4.1.

(Nós plotamos a taxa de eficiência energética, que é o inverso do consumo de energia, de modo que, assim como no ganho de velocidade, uma taxa maior seja melhor.)

A média harmônica do ganho de velocidade para o Benchmark Java é 1,28, apesar de os dois benchmarks verificarem pequenos ganhos. Esses dois benchmarks, pjbb2055 e tradebeans, embora multithreaded, têm paralelismo limitado. Eles são incluídos porque são típicos de um benchmark multithreaded que pode ser executado em um processador SMT com a esperança de extrair algum desempenho, que eles obtêm de modo limitado. Os benchmarks PARSEC obtêm ganhos de velocidade um pouco melhores do que o conjunto completo de benchmarks Java (média harmônica de 1,31). Se o tradebeans e o pjbb2005 fossem omitidos, na verdade a carga de trabalho Java teria um ganho de velocidade significativamente melhor (1,39) do que os benchmarks PARSEC. (Ver discussão sobre a implicação de usar a média harmônica para resumir os resultados na legenda da Figura 3.36.)

O consumo de energia é determinado pela combinação do ganho de energia e pelo aumento no consumo de potência. Para os benchmarks Java os SMT proporcionam, em média, a mesma eficiência energética que os não SMT (média de 1,0), mas essa eficiência é reduzida pelos dois benchmarks de desempenho ruim. Sem o tradebeans e o pjbb2005, a eficiência energética média para os benchmarks Java é de 1,06, o que é quase tão bom quanto os benchmarks PARSEC. Nos benchmarks PARSEC, o SMT reduz a energia em 1 – (1/1,08) = 7%. Tais melhorias de desempenho de redução de

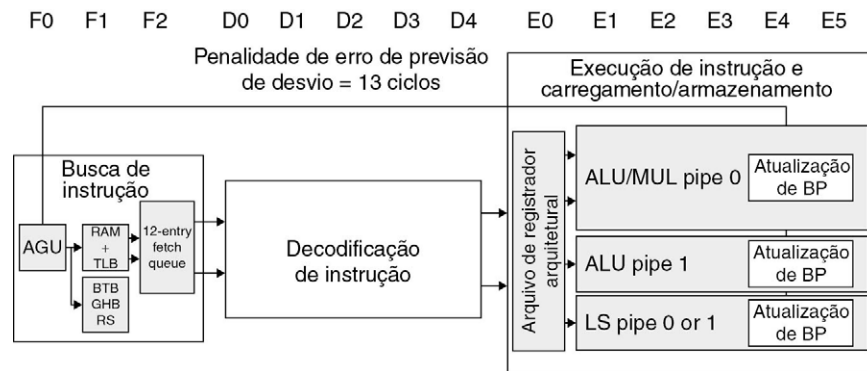



FIGURA 3.36 A estrutura básica do pipeline A8 é de 13 estágios.
São usados três ciclos para a busca de instruções e quatro para a decodificação de instruções, além de um pipeline inteiro de cinco ciclos. Isso gera uma penalidade de erro de previsão de desvio de 13 ciclos. A unidade de busca de instruções tenta manter a fila de instruções de 12 entradas cheia.

energia são *muito difíceis* de encontrar. Obviamente, a potência estática associada ao SMT é paga nos dois casos, por isso os resultados provavelmente exageram um pouco nos ganhos de energia.

Esses resultados mostram claramente que o SMT em um processador especulativo agressivo com suporte extensivo para SMT pode melhorar o desempenho em eficiência energética, o que as técnicas ILP mais agressivas não conseguiram. Em 2011, o equilíbrio entre oferecer múltiplos núcleos mais simples e menos núcleos mais sofisticados mudou em favor de mais núcleos, com cada núcleo sendo um superescalar com 3-4 despachos com SMT suportando 2-4 threads. De fato, Esmaeilzadeh *et al.* (2011) mostram que as melhorias de energia derivadas do SMT são ainda maiores no Intel i5 (um processador similar ao i7, mas com caches menores e taxa menor de clock) e o Intel Atom (um processador 80x86 projetado para o mercado de netbooks e descrito na Seção 3.14).

3.13 JUNTANDO TUDO: O INTEL CORE I7 E O ARM CORTEX-A8

Nesta seção exploraremos o projeto de dois processadores de múltiplos despachos: o ARM Cortex-A8, que é usado como base para o processador Apple A9 no iPad, além de ser o processador no Motorola Droid e nos iPhones 3GS e 4, e o Intel Core i7, um processador sofisticado especulativo, escalonado dinamicamente, voltado para desktops sofisticados e aplicações de servidor. Vamos começar com o processador mais simples.

O ARM Cortex-A8

O A8 é um superescalar com despacho duplo, escalonado estaticamente com detecção dinâmica de despacho que permite ao processador enviar uma ou duas instruções por clock. A Figura 3.36 mostra a estrutura básica do pipeline de 13 estágios.

O A8 usa um previsor dinâmico de desvio com um buffer associativo por conjunto de duas vias com 512 entradas para alvos de desvio e um buffer global de histórico de 4 K entradas, que é indexado pelo histórico de desvios e pelo PC atual. Caso o buffer de alvo de desvio erre, uma previsão será obtida do buffer global de histórico, que poderá ser usado para calcular o endereço do desvio. Além disso, uma pilha de retorno de oito entradas é mantida para rastrear os endereços de retorno. Uma previsão incorreta resulta em uma penalidade de 13 ciclos quando o pipeline é descartado.

A Figura 3.37 mostra o pipeline de decodificação de instrução. Até duas instruções por clock podem ser enviadas usando um mecanismo de despacho em ordem. Uma simples estrutura de scoreboard é usada para rastrear quando uma instrução pode ser enviada. Um par de instruções dependentes pode ser processado através da lógica de despacho, mas, obviamente, elas serão serializadas no scoreboard, a menos que possam ser enviadas para que os ganhos de avanço possam resolver a dependência.

A Figura 3.38 mostra o pipeline de execução para o processador A8. A instrução 1 ou a instrução 2 pode ir para o pipeline de load/store. O contorno total é suportado entre os pipelines. O pipeline do ARM Cortex-A8 usa um superescalar simples escalonado estaticamente de dois despachos para permitir uma frequência do clock razoavelmente alta com menor potência. Em contraste, o i7 usa uma estrutura de pipeline especulativa razoavelmente agressiva, escalonada dinamicamente com quatro despachos.

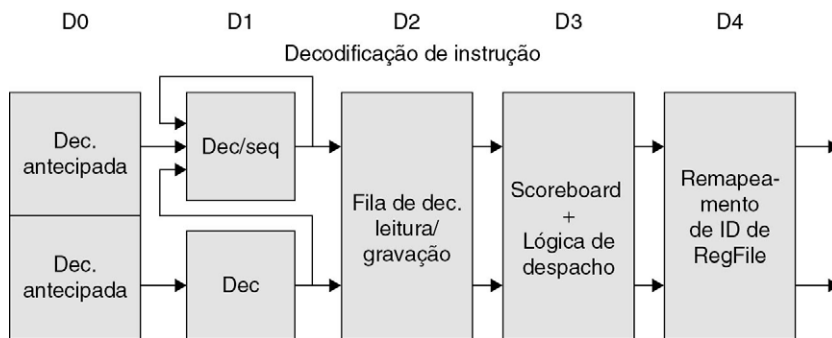



FIGURA 3.37 Decodificação de instruções em cinco estágios do A8.
No primeiro estágio, um PC produzido pela unidade de busca (seja do buffer de alvo de desvio seja do incrementador de PC) é usado para atingir um bloco de 8 bytes da cache. Até duas instruções são decodificadas e colocadas na fila de decodificação. Se nenhuma instrução for um desvio, o PC é incrementado para a próxima busca. Uma vez na fila de decodificação, a lógica de scoreboard decide quando as instruções podem ser enviadas. No despacho, os registradores operandos são lidos. Lembre-se de que em um scoreboard simples os operandos sempre vêm dos registradores. Os registradores operandos e o opcode são enviados para a parte de execução de instruções do pipeline.

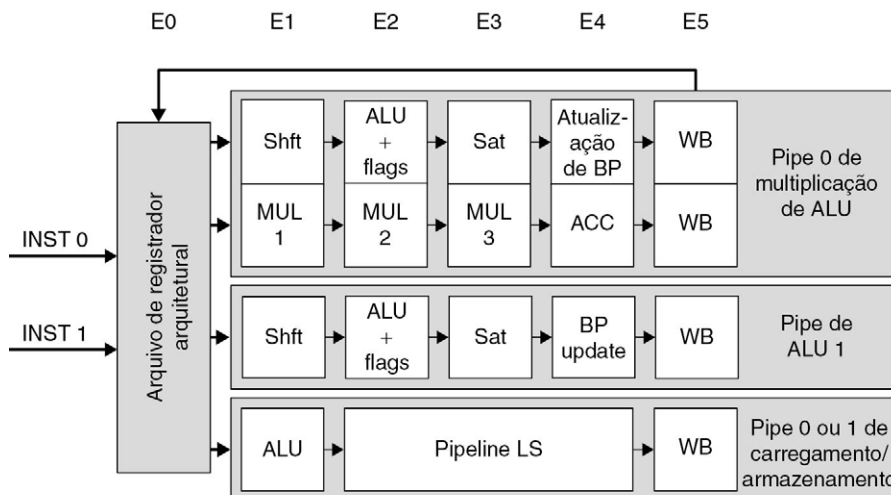



FIGURA 3.38 Decodificação de cinco estágios do A8.
Operações de multiplicação são sempre realizadas no pipeline 0 da ALU.

Desempenho do pipeline do A8

O A8 tem um CPI ideal de 0,5, devido a sua estrutura de despacho duplo. Os stalls de pipeline podem surgir de três fontes:

1. Hazards funcionais, que ocorrem porque duas instruções adjacentes selecionadas simultaneamente para despacho usam o mesmo pipeline funcional. Como o A8 é escalonado estaticamente, é tarefa do compilador tentar evitar tais conflitos. Quando eles não podem ser evitados, o A8 pode enviar, no máximo, uma instrução nesse ciclo.
2. Hazard de dados, que são detectados precocemente no pipeline e podem causar o stall das duas instruções (se a primeira não puder ser enviada, a segunda sempre sofrerá stall) ou da segunda instrução de um par. O compilador é responsável por impedir tais stalls sempre que possível.
3. Hazards de controle que surgem somente quando desvios são previstos incorretamente.

Além dos stalls de pipeline, as falhas de L1 e L2 causam stalls.

A Figura 3.39 mostra uma estimativa dos fatores que podem contribuir para o CPI real dos benchmarks Minnespec, que vimos no Capítulo 2. Como podemos ver, os atrasos

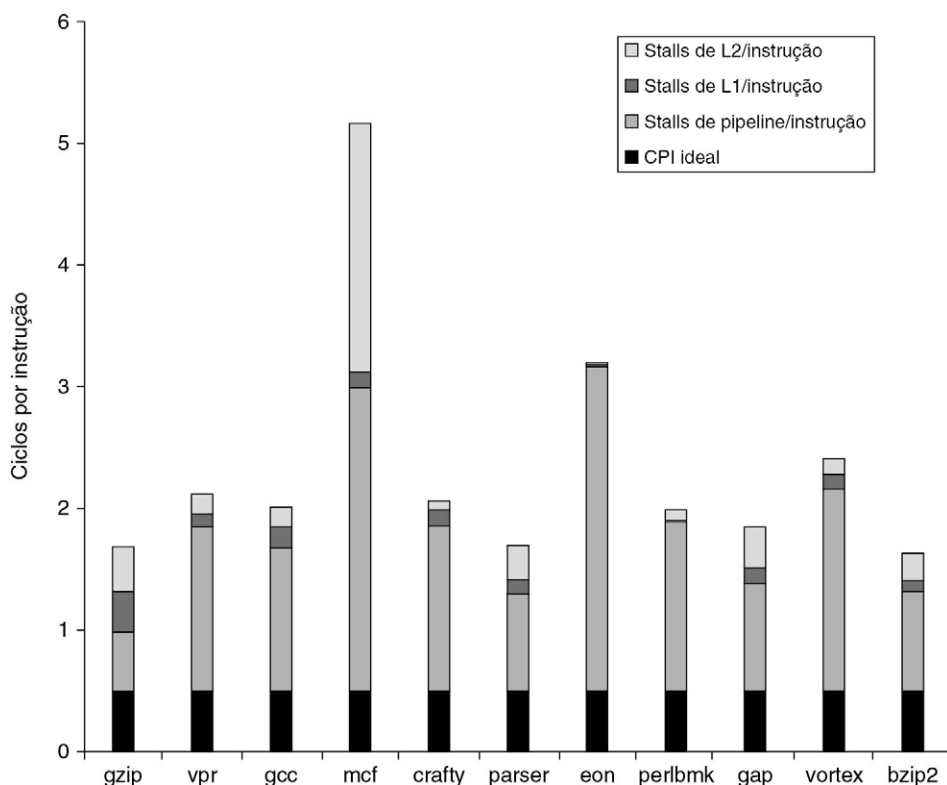



FIGURA 3.39 A composição estimada do CPI no ARM A8 mostra que os stalls de pipeline são a principal adição ao CPI base.

O eon merece menção especial, já que realiza cálculos de gráficos baseados em números inteiros (rastreamento de raio) e tem poucas falhas de cache. Ele é computacionalmente intenso, com uso pesado de multiplicação, e o único pipeline de multiplicação se torna um grande gargalo. Essa estimativa é obtida usando as taxas de falha e penalidades de L1 e L2 para calcular os stalls de L1 e L2 gerados por instrução. Estas são subtraídas do CPI medido por um simulador detalhado para obter os stalls de pipeline. Todos esses stalls incluem as três ameaças e efeitos menores, como erro de previsão de trajeto.

de pipeline, e não os stalls de memória, são os maiores contribuidores para o CPI. Esse resultado se deve parcialmente ao efeito de o Minnespec ter uma pegada menor de cache do que o SPEC completo ou outros programas grandes.

A compreensão de que os stalls de pipeline criaram perdas de desempenho significativas provavelmente teve um papel importante na decisão de fazer do ARM Cortex-A8 um superescalar escalonado dinamicamente. O A9, como o A8, envia até duas instruções por clock, mas usa escalonamento dinâmico e especulação. Até quatro instruções pendentes (duas ALUs, um load/store ou PF/multimídia e um desvio) podem começar sua execução em um ciclo de clock. O A9 usa um previsor de desvio mais poderoso, pré-fetch na cache de instrução e uma cache de dados L1 sem bloqueio. A Figura 3.40 mostra que o A9 tem desempenho melhor que o A8 por um fator de 1,28, em média, supondo a mesma frequência do clock e configurações de cache quase idênticas.

O Intel Core i7

O i7 usa uma microestrutura especulativa agressiva fora de ordem com pipelines razoavelmente profundos com o objetivo de atingir alto throughput de instruções combinando múltiplos despachos e altas taxas de clock. A Figura 3.41 mostra a estrutura geral do pipeline do i7. Vamos examinar o pipeline começando com a busca de instruções e continuando rumo à confirmação de instrução, seguindo os passos mostrados na figura.

1. Busca de instruções. O processador usa um buffer de endereços-alvo de desvio multiníveis para atingir um equilíbrio entre velocidade e precisão da previsão. Há também uma pilha de endereço de retorno para acelerar o retorno de função. Previsões

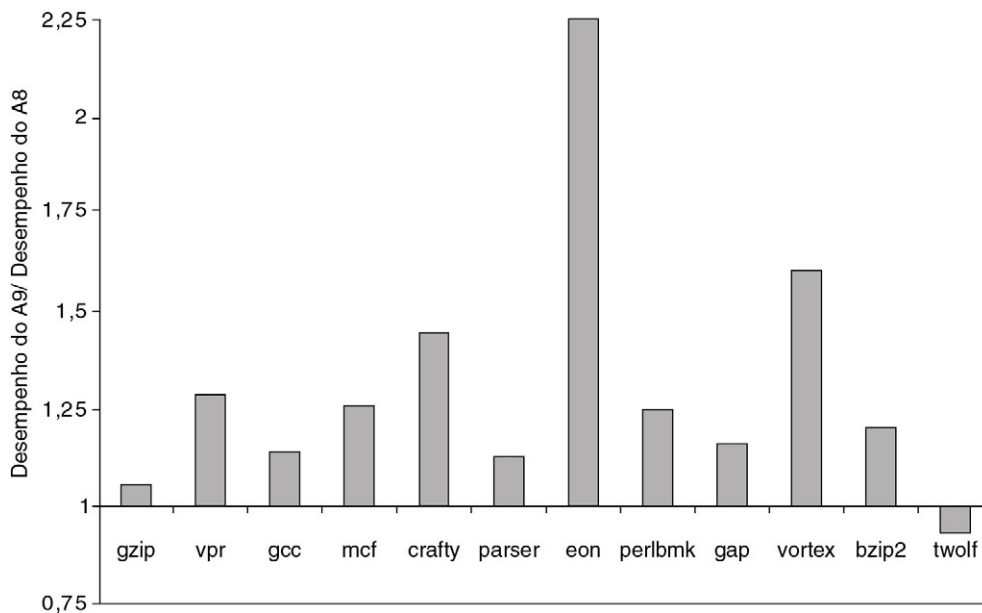



FIGURA 3.40 A taxa de desempenho do A9, comparada à do A8, ambos usando um clock de 1 GHz e os mesmos tamanhos de cache para L1 e L2, mostra que o A9 é cerca de 1,28 vez mais rápido. Ambas as execuções usam cache primária de 32 KB e cache secundária de 1 MB, que é associativo de conjunto de oito vias para o A8 e 16 vias para o A9. Os tamanhos de bloco nas caches são de 64 bytes para o A8 e de 32 bytes para o A9. Como mencionado na legenda da Figura 3.39, o eon faz uso intenso de multiplicação de inteiros, e a combinação de escalonamento dinâmico e um pipeline de multiplicação mais rápido melhora significativamente o desempenho no A9. O twolf experimenta ligeira redução de velocidade, provavelmente devido ao fato de que seu comportamento de cache é pior com o tamanho menor de bloco de L1 do A9.

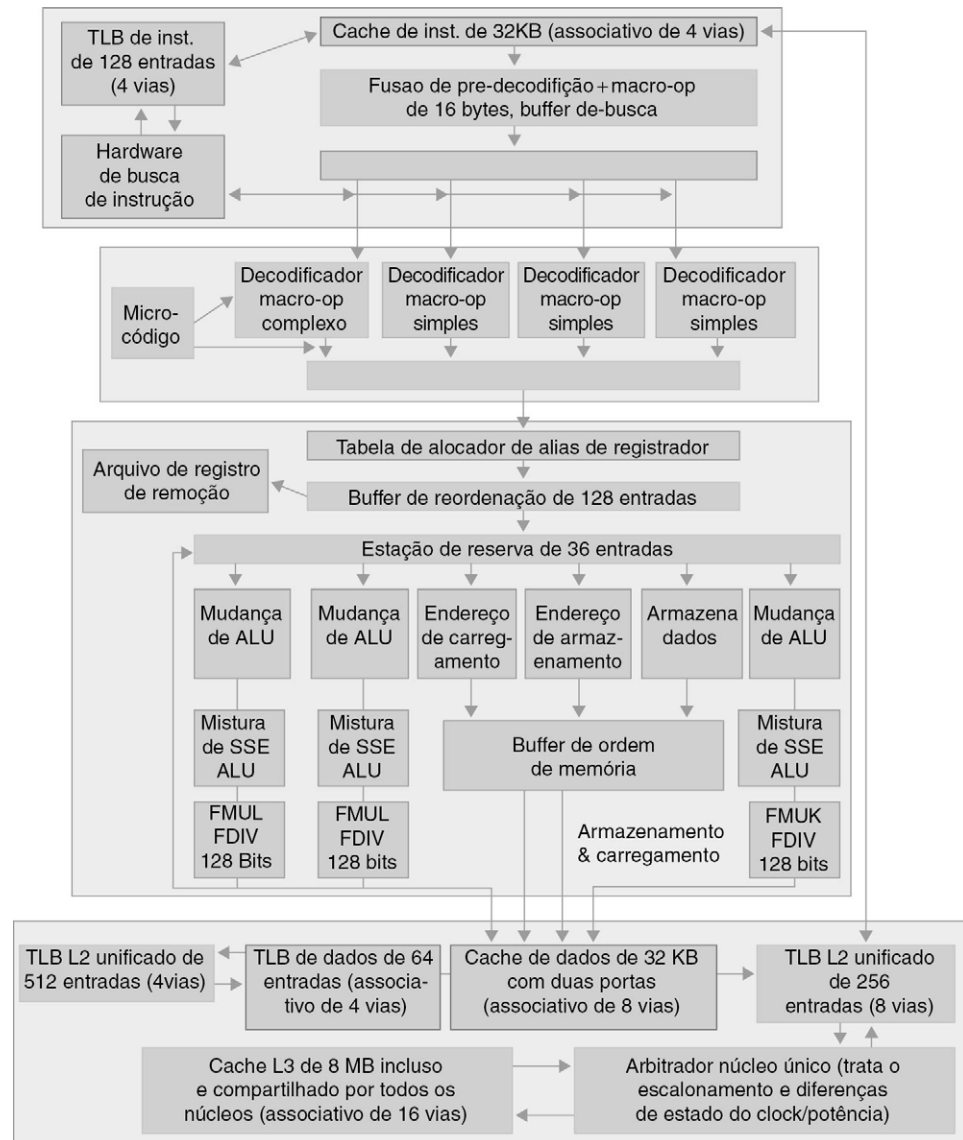



FIGURA 3.41 A estrutura de pipeline do Intel Core i7 mostrada com os componentes do sistema de memória. A profundidade total do pipeline é de 14 estágios, com erros de previsão de desvio custando 17 ciclos. Existem 48 buffers de carregamento e 32 de armazenamento. As seis unidades funcionais independentes podem começar a execução de uma micro-op no mesmo ciclo.

incorretas causam uma penalidade de cerca de 15 ciclos. Usando os endereços previstos, a unidade de busca de instrução busca 16 bytes da cache de instrução.

2. Os 16 bytes são colocados no buffer de pré-decodificação de instrução — nesse passo, um processo chamado fusão macro-op é realizado. A *fusão macro-op* toma as combinações de instrução como comparação, seguido por um desvio, e as funde em uma única operação. O estágio de pré-decodificação também quebra os 16 bytes em instruções x86 individuais. Essa pré-decodificação não é trivial, uma vez que o tamanho de uma instrução x86 pode ser de 1-17 bytes e o pré-decodificador deve examinar diversos bytes antes de saber o comprimento da instrução. Instruções x86 individuais (incluindo algumas instruções fundidas) são colocadas na fila de instruções de 18 entradas.

3. Decodificação micro-op. Instruções x86 individuais são traduzidas em micro-ops. Micro-ops são instruções simples, similares às do MIPS, que podem ser executadas diretamente pelo pipeline. Essa técnica de traduzir o conjunto de instruções x86 em operações simples mais fáceis de usar em pipeline foi introduzida no Pentium Pro em 1997 e tem sido usada desde então. Três dos decodificadores tratam instruções x86 que as traduzem diretamente em uma micro-op. Para instruções x86 que têm semântica mais complexa, existe um máquina de microcódigo que é usada para produzir a sequência de micro-ops. Ela pode produzir até quatro micro-ops a cada ciclo e continua até que a sequência de micro-ops necessária tenha sido gerada. As micro-ops são posicionadas de acordo com a ordem das instruções x86 no buffer de micro-ops de 28 entradas.
4. O buffer de micro-op realiza *detecção* e *microfusão*. Se houver uma sequência pequena de instruções (menos de 28 instruções ou 256 bytes de comprimento) que contenham um loop, o detector de fluxo de loop vai encontrar o loop e enviar diretamente as micro-ops do buffer, eliminando a necessidade de ativar os estágios de busca de instrução e decodificação da instrução. A microfusão combina pares de instruções, como load/operação ALU e a operação ALU/store, e as envia para uma única estação de reserva (onde elas ainda podem ser enviadas independentemente), aumentando assim o uso do buffer. Em um estudo da arquitetura do Intel Core, que também incorpora microfusão e macrofusão, Bird *et al.* (2007) descobriram que a microfusão tinha pouco impacto no desempenho, enquanto a macrofusão parece ter um impacto positivo modesto no desempenho com inteiros e pouco impacto sobre o desempenho com ponto flutuante.
5. Realizar o despacho da instrução básica. Buscar a localização do registrador nas tabelas de registro, renomear os registradores, alocar uma entrada no buffer de reordenação e buscar quaisquer resultados dos registradores ou do buffer de reordenação antes de enviar as micro-ops para as estações de reserva.
6. O i7 usa uma estação de reserva de 36 entradas centralizada compartilhada por seis unidades funcionais. Até seis micro-ops podem ser enviadas para as unidades funcionais a cada ciclo de clock.
7. As micro-ops são executadas pelas unidades funcionais individuais e então os resultados são enviados de volta para qualquer estação de reserva, além da unidade de remoção de registrador, onde elas vão atualizar o status do registrador, uma vez que se saiba que a instrução não é mais especulativa. A entrada correspondente à instrução no buffer de reordenação é marcada como completa.
8. Quando uma ou mais instruções no início do buffer de reordenação são marcadas como completas, as gravações pendentes na unidade de remoção de registrador são executadas e as instruções são removidas do buffer de reordenação.

Desempenho do i7

Em seções anteriores, nós examinamos o desempenho do previsor de desvio do i7 e também o desempenho do SMT. Nesta seção, examinaremos o desempenho do pipeline de thread único. Por causa da presença de especulação agressiva e de caches sem bloqueio, é difícil avaliar com precisão a distância entre o desempenho idealizado e o desempenho real. Como veremos, poucos stalls ocorrem, porque as instruções não podem enviar. Por exemplo, somente cerca de 3% dos loads são atrasados, porque nenhuma estação de reserva está disponível. A maioria das falhas vem de previsões incorretas de desvio ou de falhas de cache. O custo de uma previsão incorreta de desvio é de 15 ciclos, enquanto o custo de uma falha de L1 é de cerca de 10 ciclos. As falhas de L2 são cerca de três vezes mais caras do que uma falha de L1, e as falhas de L3 custam cerca de 13 vezes o custo de uma falha de L1 (130-135 ciclos)! Embora o processador tente encontrar instruções

alternativas para executar para falhas de L3 e algumas falhas de L2, é provável que alguns dos buffers sejam preenchidos antes de a falha se completar, fazendo o processador parar de enviar instruções.

Para examinar o custo de previsões e especulações incorretas, a Figura 3.42 mostra a fração do trabalho (medida pelos números de micro-ops enviadas para o pipeline) que não são removidas (ou seja, seus resultados são anulados) em relação a todos os despachos de micro-op. Para o sjeng, por exemplo, 25% do trabalho é desperdiçado, já que 25% das micro-ops enviadas nunca são removidas.

Observe que, em alguns casos, o trabalho perdido é muito próximo das taxas de previsão incorreta mostradas na Figura 3.5, na página 144, mas em vários outros, como o mcf, o trabalho perdido parece relativamente maior do que a taxa de previsão incorreta. Em tais casos, uma explicação provável está no comportamento da memória. Com as taxas muito altas de falha de cache de dados, o mcf vai enviar muitas instruções durante uma especulação incorreta enquanto houver estações de reserva suficientes disponíveis para as referências de memória que sofreram stall. Quando a previsão incorreta de desvio é detectada, as micro-ops correspondentes a essas instruções são descartadas, mas ocorre um congestionamento nas caches, conforme as referências especuladas à memória tentam ser completadas. Não há modo simples de o processador interromper tais requisições de cache depois que elas são iniciadas.

A Figura 3.43 mostra o CPI geral para os 19 benchmarks SPEC CPU2006. Os benchmarks inteiros têm um CPI de 1,06 com variância muito grande (0,67 de desvio-padrão). O MCF e o OMNETTP são as principais discrepâncias, ambos tendo um CPI de mais de 2,0, enquanto a maioria dos outros benchmarks estão próximos de 1,0 ou são menores do que isso (o gcc, o segundo maior, tem 1,23). Essa variância é decorrente de diferenças na precisão da previsão de desvio e nas taxas de falha de cache. Para os benchmarks inteiros,

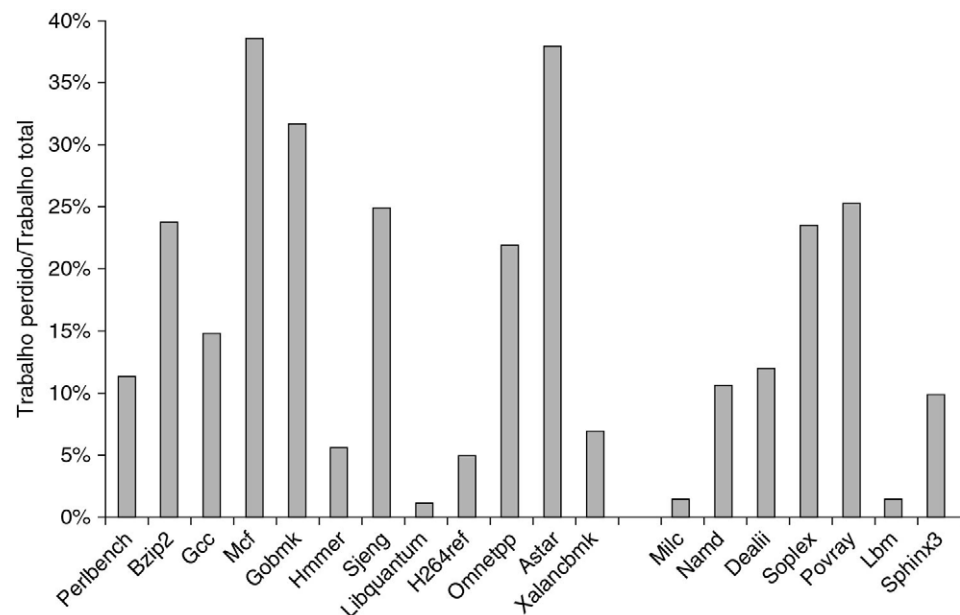



FIGURA 3.42 A quantidade de "trabalho perdido" é plotada tomando a razão entre micro-ops enviadas que não são graduadas e todas as micro-ops enviadas.

Por exemplo, a razão é de 25% para o sjeng, significando que 25% das micro-ops enviadas e executadas são jogadas fora. Os dados apresentados nesta seção foram coletados pelo professor Lu Peng e pelo doutorando Ying Zhang, ambos da Universidade do Estado da Louisiana.

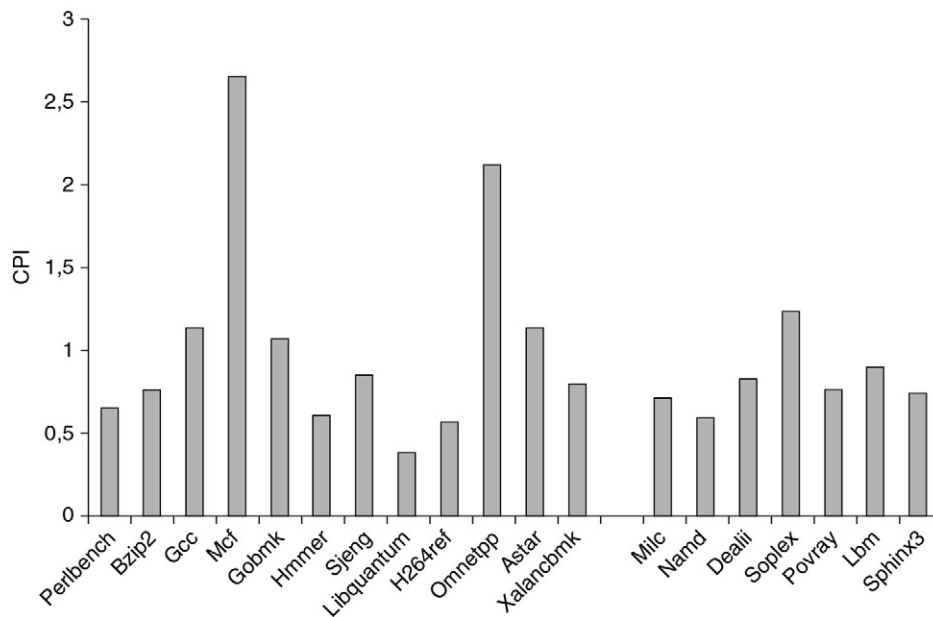



FIGURA 3.43 **O CPI para os 19 benchmarks SPECCPU2006 mostra um CPI médio de 0,83 para os benchmarks PF e de inteiro, embora o comportamento seja bastante diferente.**
No caso dos inteiros, os valores de CPI variam de 0,44-2,66, com desvio-padrão de 0,77, enquanto no caso de PF a variação é de 0,62-1,38, com desvio-padrão de 0,25. Os dados nesta seção foram coletados pelo professor Lu Peng e pelo doutorando Ying Zhang, ambos da Universidade do Estado da Louisiana.

a taxa de falha de L2 é o melhor previsor de CPI, e a taxa de falhas de L3 (que é muito pequeno) praticamente não tem efeito.

Os benchmarks de PF atingem desempenho maior com CPI médio menor (0,89) e desvio-padrão menor (0,25). Para os benchmarks de PF, L1 e L2 são igualmente importantes para determinar o CPI, enquanto o L3 tem um papel menor mais significativo. Embora o escalonamento dinâmico e as capacidades de não bloqueio do i7 possam ocultar alguma latência de falha, o comportamento da memória de cache ainda é responsável por uma grande contribuição. Isso reforça o papel do multithreading como outro modo de ocultar a latência de memória.

3.14 FALÁCIAS E ARMADILHAS

Nossas poucas falácias se concentram na dificuldade de prever o desempenho e a eficiência energética e de extrapolar medidas únicas, como frequência do clock ou CPI. Mostraremos também que diferentes técnicas de arquiteturas podem ter comportamentos radicalmente diferentes para diferentes benchmarks.

Falácia. *É fácil prever o desempenho e a eficiência energética de duas versões diferentes da mesma arquitetura de conjunto de instruções, se mantivermos a tecnologia constante.*

A Intel fabrica um processador para a utilização final em netbooks e PDMs que é muito similar ao ARM A8 na sua microarquitetura: é o chamado Atom 230. É interessante que o Atom 230 e o Core i7 920 tenham sido fabricados com a mesma tecnologia de 45 nm da Intel. A Figura 3.44 resume o Intel Core i7, o ARM Cortex-A8 e o Intel Atom 230. Essas similaridades proporcionam uma rara oportunidade de comparar diretamente duas microarquiteturas radicalmente diferentes para o mesmo conjunto de instruções e, ao mesmo

		Intel i7 920	ARM A8	Intel Atom 230
Área	**Característica específica**	**Quatro núcleos, cada um com PF**	**Um núcleo, sem PF**	**Um núcleo, com PF**
	Frequência do clock	2,66 GHz	1 GHz	1,66 GHz
Propriedades físicas	Potência térmica de projeto	130 W	2 W	4 W
	Encapsulamento	BGA de 1.366 pinos	BGA de 522 pinos	BGA de 437 pinos
	TLB	L2 128 I/ 64D 512 com dois níveis, todos associativos por conjunto de 4 vias	32 I/ 32D de um nível totalmente associativo	L2 16 I/ 16D 64 com dois níveis, todos associativos por conjunto de 4 vias
Sistema de memória	Caches	32 KB/32 KB 256 KB 2-8 MB de três níveis	16/16 ou 32/32 KB 128 KB-1 MB de dois níveis	32/24 KB 512 KB de dois níveis
	BW de pico de memória	17 GB/s	12 GB/s	8 GB/s
	Taxa pico de despacho	4 ops/clock com fusão	2 ops/clock	2 ops/clock
Estrutura de pipeline	Escalonamento de pipeline	Especulação fora de ordem	Despacho dinâmico em ordem	Despacho dinâmico em ordem
	Previsão de desvio	Dois níveis	BTB de 512 entradas de dois níveis, histórico global de 4 K pilha de retorno de 8 entradas	Dois níveis

FIGURA 3.44 Visão geral do Intel i7 920 de quatro núcleos, exemplo de um processador ARM A8 (com um L2 de 256 MB, L1 de 32 KB e sem ponto flutuante) e o Intel ARM 230 mostrando claramente a diferença em termos de filosofia de projeto entre um processador voltado para PMD (no caso do ARM) ou espaço de netbook (no caso do Atom) e um processador para uso em servidores e desktops sofisticados. Lembre-se de que o i7 inclui quatro núcleos, cada qual várias vezes mais alto em desempenho do que o A8 ou o Atom de um núcleo. Todos esses processadores são implementados em uma tecnologia comparável de 45 nm.

tempo, manter constante a tecnologia de fabricação fundamental. Antes de fazermos a comparação, precisamos contar um pouco mais sobre o Atom 230.

Os processadores Atom implementam a arquitetura x86 em instruções similares às RISC (como toda implementação x86 tem feito desde meados dos anos 1990). O Atom usa uma micro-operação ligeiramente mais poderosa, que permite que uma operação aritmética seja pareada com um carregamento ou armazenamento. Isso significa que, em média, para um *mix* típico de instruções, somente 4% das instruções requerem mais de uma micro-operação. As micro-operações são executadas em um pipeline com profundidade de 16, capaz de enviar duas instruções por clock, em ordem, como no ARM A8. Há duas ALUs de inteiro duplo, pipelines separados para soma de PF e outras operações de PF, e dois pipelines de operação de memória suportando execução dupla, mais geral do que o ARM A8, porém ainda limitadas pela capacidade de despacho em ordem. O Atom 230 tem cache de instrução de 32 KB e cache de dados de 24 KB, ambas suportadas por um L2 de 512 KB no mesmo substrato. (O Atom 230 também suporta multithreading com dois threads, mas vamos considerar somente comparações de um único thread.) A Figura 3.46 resume os processadores i7, A8 e Atom e suas principais características.

Podemos esperar que esses dois processadores, implementados na mesma tecnologia e com o mesmo conjunto de instruções, apresentassem comportamento previsível em termos de desempenho relativo e consumo de energia, o que significa que a potência e o desempenho teriam uma escala próxima à linearidade. Examinamos essa hipótese usando três conjuntos de benchmarks. Os primeiros conjuntos são um grupo de benchmarks Java de thread único que vêm dos benchmarks DaCapo, e dos benchmarks SPEC JVM98 (ver discussão sobre os benchmarks e medidas em Esmaeilzadeh *et al.*, 2011). O segundo e o terceiro conjuntos são do SPEC CPU2006 e consistem, respectivamente, nos benchmarks para inteiros e PF.

Como podemos ver na Figura 3.45, o i7 tem desempenho significativamente maior do que o do Atom. Todos os benchmarks são pelo menos quatro vezes mais rápidos no i7, dois benchmarks SPECFP são mais de 10 vezes mais rápidos, e um benchmark SPECINT é executado mais de oito vezes mais rápido!

Como a razão das taxas de clock desses dois processadores é de 1,6, a maior parte da vantagem vem de um CPI muito menor para o i7: um fator de 2,8 para os benchmarks Java, um fator de 3,1 para os benchmarks SPECINT e um fator de 4,3 para os benchmarks SPECFP.

Mas o consumo médio de energia para o i7 está pouco abaixo de 43 W, enquanto o consumo médio de energia do Atom é de 4,2 W, ou cerca de um décimo da energia! Combinar o desempenho e a energia leva a uma vantagem na eficiência energética para o Atom, que

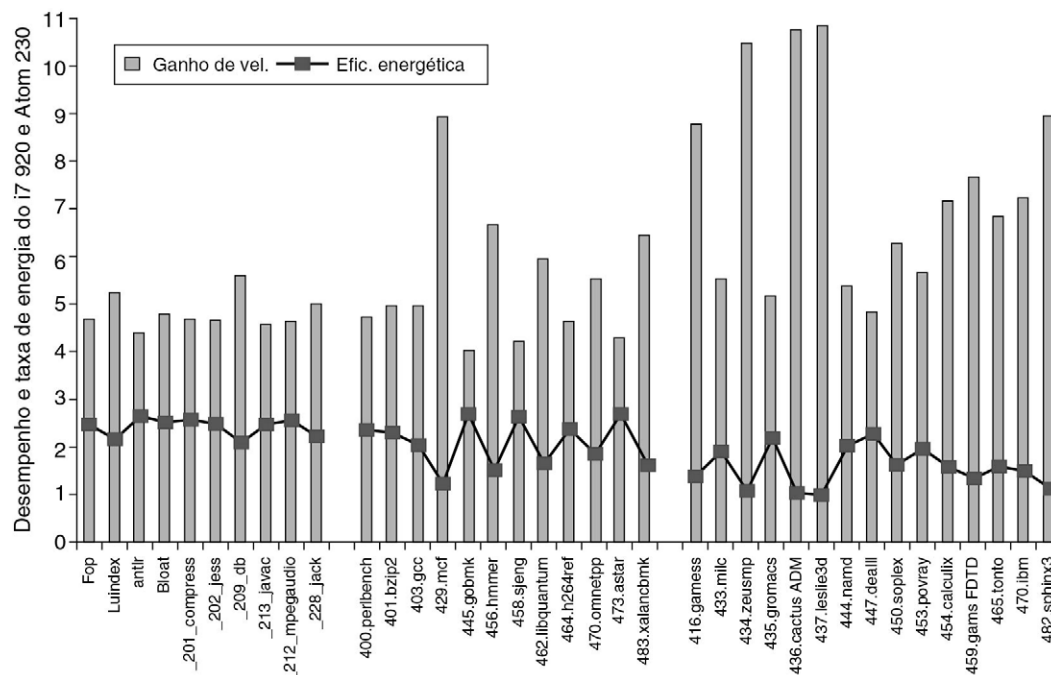



FIGURA 3.45 O desempenho relativo e a eficiência energética para um conjunto de benchmarks de thread único mostram que o i7 920 é 4-10 vezes mais rápido do que o Atom 230, porém cerca de duas vezes menos eficiente em termos de potência, em média!
O desempenho é mostrado nas colunas como o i7 em relação ao Atom, como tempo de execução (i7)/tempo de execução (Atom). A energia é mostrada pela linha como energia(Atom)/energia(i7). O i7 nunca vence o Atom em eficiência energética, embora seja essencialmente tão bom em quatro benchmarks, três dos quais são de ponto flutuante. Os dados mostrados aqui foram coletados por Esmaelizadeh *et al.* (2011). Os benchmarks SPEC foram compilados com otimização sobre o uso do compilador-padrão Intel, enquanto os benchmarks Java usam o Sun (Oracle) Hotspot Java VM. Somente um núcleo está ativo no i7, e o resto está em modo de economia de energia profunda. O Turbo Boost é usado no i7, que aumenta sua vantagem de desempenho mas diminui levemente sua eficiência energética relativa.

costuma ser mais de 1,5-2 vezes melhor! Essa comparação de dois processadores usando a mesma tecnologia fundamental torna claro que as vantagens do desempenho de um superescalar agressivo com escalonamento dinâmico e especulação vêm com significativa desvantagem em termos de eficiência energética.

Falácia. *Processadores com CPIs menores sempre serão mais rápidos.*

Falácia. *Processadores com taxas de clock mais rápidas sempre serão mais rápidos.*

A chave é que o produto da CPI e a frequência do clock determinam o desempenho. Com alta frequência do clock obtida por um pipelining longo, a CPU deve manter um CPI baixo para obter o benefício total do clock mais rápido. De modo similar, um processador simples com alta frequência do clock mas CPI baixo pode ser mais lento.

Como vimos na falácia anterior, o desempenho e a eficiência energética podem divergir significativamente entre processadores projetados para ambientes diferentes, mesmo quando eles têm o mesmo ISA. Na verdade, grandes diferenças em desempenho podem aparecer até dentro de uma família de processadores da mesma companhia, projetados todos para aplicações de alto nível. A Figura 3.46 mostra o desempenho para inteiros e PF de duas implementações diferentes da arquitetura x86 da Intel, além de uma versão da arquitetura Itanium, também da Intel.

O Pentium 4 foi o processador com pipeline mais agressivo já construído pela Intel. Ele usava um pipeline com mais de 20 estágios, tinha sete unidades funcionais e micro-ops em cache no lugar de instruções x86. Seu desempenho relativamente inferior, dada a implementação agressiva, foi uma indicação clara de que a tentativa de explorar mais ILP (podia haver facilmente 50 operações em ação) havia falhado. O consumo de energia do Pentium era similar ao do i7, embora sua contagem de transistores fosse menor, já que as caches primárias tinham metade do tamanho das do i7, e incluía somente uma cache secundária de 2 MB, sem cache terciária.

O Intel Itanium é uma arquitetura no estilo VLIW que, apesar da redução potencial em complexidade em comparação aos superescalares escalonados dinamicamente, nunca atingiu taxas de clock competitivas *versus* os processadores x86 da linha principal (embora ele pareça alcançar um CPI geral similar ao do i7). Ao examinar esses resultados, o leitor deve ter em mente que eles usam diferentes tecnologias de implementação, o que dá ao i7 uma vantagem em termos de velocidade de transistor, portando frequência do clock para um processador com pipeline equivalente. Mesmo assim, a grande variação no desempenho — mais de três vezes entre o Pentium e o i7 — é surpreendente. A próxima armadilha explica de onde vem uma significativa parte dessa vantagem.

Armadilha. *Às vezes maior e mais "burro" é melhor.*

No início dos anos 2000, grande parte da atenção estava voltada para a construção de processadores agressivos na exploração de ILP, incluindo a arquitetura Pentium 4, que

Processador	Frequência do Clock	Base SPECCInt2006	Baseline SPECCFP2006
Intel Pentium 4 670	3,8 GHz	11,5	12,2
Intel Itanium-2	1,66 GHz	14,5	17,3
Intel i7	3,3 GHz	35,5	38,4

FIGURA 3.46 **Três diferentes processadores da Intel variam muito.**
Embora o processador Itanium tenha dois núcleos e o i7 tenha quatro, somente um núcleo é usado nesses benchmarks.

usava o pipeline mais longo já visto em um microprocessador, e o Intel Itanium, que tinha a mais alta taxa de pico de despacho por clock já vista. O que se tornou rapidamente claro foi que, muitas vezes, a maior limitação em explorar ILP era o sistema de memória. Embora pipelines especulativos fora de ordem fossem razoavelmente bons em ocultar uma fração significativa das penalidades de falha, 10-15 ciclos para uma falha de primeiro nível, muitas vezes eles faziam muito pouco para ocultar as penalidades para uma falha de segundo nível, que, quando ia para a memória principal, provavelmente era de 50-100 ciclos de clock.

O resultado foi que esses projetos nunca chegaram perto de atingir o pico de throughput de instruções, apesar do grande número de transistores e das técnicas extremamente sofisticadas e inteligentes. A próxima seção discute esse dilema e o afastamento dos esquemas de ILP mais agressivos em favor dos núcleos múltiplos, mas houve outra mudança, que é um exemplo dessa armadilha. Em vez de tentar ocultar ainda mais latências de memória com ILP, os projetistas simplesmente usaram os transistores para construir caches muito maiores. O Itanium 2 e o i7 usam caches de três níveis em comparação à cache de dois níveis do Pentium 4. Não é necessário dizer que construir caches maiores é muito mais fácil do que projetar o pipeline com mais de 20 estágios do Pentium 4 e, a partir dos dados da Figura 3.46, parece ser também mais eficaz.

3.15 COMENTÁRIOS FINAIS: O QUE TEMOS À FRENTE?

No início de 2000, o foco era explorar o paralelismo em nível de instrução. A Intel estava prestes a lançar o Itanium, processador escalonado estaticamente com alta taxa de despacho que contava com uma técnica semelhante à VLIW, com suporte intensivo do compilador. Os processadores MIPS, Alpha e IBM, com execução especulativa escalonada dinamicamente, estavam na segunda geração e se tornando maiores e mais rápidos. O Pentium 4, que usava escalonamento especulativo, também havia sido anunciado naquele ano com sete unidades funcionais e um pipeline com mais de 20 estágios de comprimento. Mas havia algumas nuvens pesadas no horizonte.

Pesquisas como a abordada na Seção 3.10 mostravam que levar o ILP muito adiante seria extremamente difícil, e, embora as taxas de throughput de pico de instruções tivessem aumentado em relação aos primeiros processadores especulativos de 3-5 anos antes, as taxas sustentáveis de execução de instrução estavam aumentando muito mais lentamente.

Os cinco anos seguintes disseram muito. O Itanium acabou sendo um bom processador de PF, mas apenas um processador de inteiros medíocre. A Intel ainda produz a linha, mas não há muitos usuários; a frequência do clock está aquém da frequência dos processadores da linha principal da Intel, e a Microsoft não suporta mais o conjunto de instruções. O Intel Pentium 4, embora tivesse bom desempenho, acabou sendo ineficiente em termos de desempenho/watt (ou seja, uso de energia), e a complexidade do processador tornou improvável que mais avanços fossem possíveis aumentando a taxa de despacho. Havia chegado o fim de uma estrada de 20 anos atingindo novos níveis de desempenho em microprocessadores explorando o ILP. O Pentium 4 foi amplamente reconhecido como tendo ido além do ponto, e a agressiva e sofisticada microarquitetura Netburst foi abandonada.

Em 2005, a Intel e os demais fabricantes principais haviam renovado sua abordagem para se concentrar em núcleos múltiplos. Um desempenho maior seria alcançado através do paralelismo em nível de thread, em vez de paralelismo em nível de instrução, e a responsabilidade por usar o processador com eficiência mudaria bastante do hardware para o software e para o programador. Essa mudança foi a mais significativa na arquitetura de processador desde os primeiros dias do pipelining e do paralelismo em nível de instrução, cerca de 25 anos antes.

No mesmo período, os projetistas começaram a explorar o uso de um paralelismo em nível de dados como outra abordagem para obter desempenho. As extensões SIMD permitiram aos microprocessadores de desktops e servidores atingir aumentos moderados de desempenho para gráficos e funções similares. E o que é mais importante: as GPUs buscaram o uso agressivo de SIMD, atingindo vantagens significativas de desempenho para aplicações com grande paralelismo em nível de dados. Para aplicações científicas, tais técnicas representam uma alternativa viável para o mais geral — porém menos eficiente — paralelismo em nível de thread explorado nos núcleos múltiplos. O Capítulo 4 vai explorar esses desenvolvimentos no uso de paralelismo em nível de dados.

Muitos pesquisadores anteviram uma grande redução no uso de ILP, prevendo que processadores superescalares com dois despachos e números maiores de núcleos seriam o futuro. Entretanto, as vantagens de taxas de despacho ligeiramente maiores e a capacidade de escalonamento dinâmico especulativo para lidar com eventos imprevisíveis, como falhas de cache de primeiro nível, levaram um ILP moderado a ser o principal bloco de construção nos projetos de núcleos múltiplos. A adição do SMT e a sua eficácia (tanto em desempenho quanto em eficiência energética) concretizaram ainda mais a posição das técnicas especulativas, fora de ordem, de despacho moderado. De fato, mesmo no mercado de embarcados, os processadores mais novos (como o ARM Cortex-A9) introduziram escalonamento dinâmico, especulação e taxas maiores de despacho.

É muito improvável que os futuros processadores tentem melhorar significativamente a largura de despacho. É simplesmente muito ineficiente, tanto do ponto de vista da utilização de silício quanto da eficiência energética. Considere os dados da Figura 3.47, que mostra os quatro processadores mais recentes da série IBM Power. Ao longo da década passada, houve uma modesta melhoria no suporte a ILP nos processadores Power, mas a parte dominante do aumento no número de transistores (um fator de quase 7 do Power4 para o Power7), aumentou as caches e o número de núcleos por die. Até mesmo a expansão no suporte a SMT parece ser mais um foco do que um aumento no throughput de ILP. A estrutura ILP do Power4 para o Power7 foi de cinco despachos para seis, de oito unidades funcionais para 12 (mas sem aumento das duas unidades de carregamento/armazenamento originais), enquanto o suporte a SMT foi de não existente para quatro threads/processador. Parece claro que, mesmo para o processador ILP mais avançado em 2011 (o Power7), o foco foi deslocado para além do paralelismo em nível de instrução.

	Power4	Power5	Power6	Power7
Lançado em	2001	2004	2007	2010
Frequência inicial do clock (GHz)	1,3	1,9	4,7	3,6
Número de transistores (M)	174	276	790	1.200
Despachos por clock	5	5	7	6
Unidades funcionais	8	8	9	12
Núcleos/chip	2	2	2	8
Threads SMT	0	2	2	4
Cache total no chip (MB)	1,5	2	4,1	32,3

FIGURA 3.47 **Características de quatro processadores IBM Power.**

Todos foram escalonados dinamicamente, exceto o Power6, que é estático, e em ordem, e todos os processadores suportam dois pipelines de carregamento/armazenamento. O Power6 tem as mesmas unidades funcionais que o Power5, exceto por uma unidade decimal. O Power7 usa DRAM para a cache L3.

Os Capítulos 4 e 5 enfocam técnicas que exploram o paralelismo em nível de dados e o paralelismo em nível de thread.

3.16 PERSPECTIVAS HISTÓRICAS E REFERÊNCIAS

A Seção L.5 (disponível on-line) contém uma análise sobre o desenvolvimento do pipelining e do paralelismo em nível de instrução. Apresentamos diversas referências para leitura adicional e exploração desses tópicos. A Seção L.5 cobre o Capítulo 3 e o Apêndice H.

ESTUDOS DE CASO E EXERCÍCIOS POR JASON D. BAKOS E ROBERT P. COLWELL

Estudo de caso: explorando o impacto das técnicas de microarquiteturas

Conceitos ilustrados por este estudo de caso

- Escalonamento básico de instrução, reordenação, despacho
- Múltiplo despacho e hazards
- Renomeação de registrador
- Execução fora de ordem e especulativa
- Onde gastar recursos fora de ordem

Você está encarregado de projetar uma nova microarquitetura de processador e tentando descobrir como alocar melhor seus recursos de hardware. Quais das técnicas de hardware e software aprendidas neste capítulo deverá aplicar? Você tem uma lista de latências para as unidades funcionais e para a memória, além de algum código representativo. Seu chefe foi um tanto vago com relação aos requisitos de desempenho do seu novo projeto, mas você sabe, por experiência, que, com tudo o mais sendo igual, mais rápido geralmente é melhor. Comece com o básico. A Figura 3.48 apresenta uma sequência de instruções e a lista de latências.

3.1 [10] <1.8, 3.1, 3.2> Qual seria o desempenho de referência (em ciclos, por iteração do loop) da sequência de código da Figura 3.48 se nenhuma nova execução de instrução pudesse ser iniciada até que a execução da instrução anterior tivesse sido concluída? Ignore a busca e a decodificação de front-end.

			Latências além de um único ciclo	
Loop:	LD	F2,0(RX)	Memory LD	+4
I0:	DIVD	F8,F2,F0	Memory SD	+1
I1:	MULTD	F2,F6,F2	ADD, SUB de inteiro	+0
I2:	LD	F4,0(Ry)	Desvios	+1
I3:	ADDD	F4,F0,F4	ADDD	+1
I4:	ADDD	F10,F8,F2	MULTD	+5
I5:	ADDI	Rx,Rx,#8	DIVD	+12
I6:	ADDI	Ry,Ry,#8		
I7:	SD	F4,0(Ry)		
I8:	SUB	R20,R4,Rx		
I9:	BNZ	R20,Loop		

FIGURA 3.48 Código e latências para os Exercícios 3.1 a 3.6.

Considere, por enquanto, que a execução não fique em stall por falta da próxima instrução, mas somente uma instrução/ciclo pode ser enviada. Considere que o desvio é tomado e que existe um slot de atraso de desvio de um ciclo.

3.2 [10] <1.8, 3.1, 3.2> Pense no que realmente significam os números de latência — eles indicam o número de ciclos que determinada função exige para produzir sua saída, e nada mais. Se o pipeline ocasionar stalls para os ciclos de latência de cada unidade funcional, pelo menos você terá a garantia de que qualquer par de instruções de ponta a ponta (um "produtor" seguido por um "consumidor") será executado corretamente. Contudo, nem todos os pares de instruções possuem um relacionamento produtor/consumidor. Às vezes, duas instruções adjacentes não têm nada a ver uma com a outra. Quantos ciclos o corpo do loop na sequência de código da Figura 3.48 exigiria se o pipeline detectasse verdadeiras dependências de dados e só elas ficassem em stall, em vez de tudo ficar cegamente em stall só porque uma unidade funcional está ocupada? Mostre o código com <stall> inserido onde for necessário para acomodar as latências proteladas. (*Dica:* Uma instrução com latência " + 2" precisa que dois ciclos de <stall> sejam inseridos na sequência de código. Pense desta maneira: uma instrução de um ciclo possui latência 1 + 0, significando zero estado de espera extra. Assim, a latência 1 + 1 implica um ciclo de stall; latência 1 + *N* possui *N* ciclos de stall extras.)

3.3 [15] <3.6, 3.7> Considere um projeto de múltiplo despacho. Suponha que você tenha dois pipelines de execução, cada qual capaz de iniciar a execução de uma instrução por ciclo, além de largura de banda de busca/decodificação suficiente no front-end, de modo que sua execução não estará em stall. Considere que os resultados podem ser encaminhados imediatamente de uma unidade de execução para outra ou para si mesma. Considere, ainda, que o único motivo para um pipeline de execução protelar é observar uma dependência de dados verdadeira. Quantos ciclos o loop exigiria?

3.4 [10] <3.6, 3.7> No projeto de múltiplo despacho do Exercício 3.3, você pode ter reconhecido algumas questões sutis. Embora os dois pipelines tenham exatamente o mesmo repertório de instruções, elas não são idênticas nem intercambiáveis, pois existe uma ordenação implícita entre elas que precisa refletir a ordenação das instruções no programa original. Se a instrução *N* + 1 iniciar sua execução na pipe de execução 1 ao mesmo tempo que a instrução *N* iniciar na pipe 0, e *N* + 1 exigir uma latência de execução mais curta que *N*, então *N* + 1 será concluída antes de *N* (embora a ordenação do programa tivesse indicado de outra forma). Cite pelo menos duas razões pelas quais isso poderia ser arriscado e exigiria considerações especiais na microarquitetura. Dê um exemplo de duas instruções do código da Figura 3.48 que demonstrem esse hazard.

3.5 [20] <3.7> Reordene as instruções para melhorar o desempenho do código da Figura 3.48. Considere a máquina de dois pipelines do Exercício 3.3 e que os problemas de término fora de ordem do Exercício 3.4 foram tratados com sucesso. Por enquanto, preocupe-se apenas em observar as dependências de dados verdadeiras e as latências da unidade funcional. Quantos ciclos o seu código reordenado utiliza?

3.6 3.6 [10/10/10] <3.1, 3.2> Cada ciclo que não inicia uma nova operação em um pipeline é uma oportunidade perdida, no sentido de que seu hardware não está "acompanhando seu potencial".

a. [10] <3.1, 3.2> Em seu código reordenado do Exercício 3.5, que fração de todos os ciclos, contando ambos os pipelines, foi desperdiçada (não iniciou uma nova operação)?

b. [10] <3.1, 3.2> O desdobramento de loop é uma técnica-padrão do compilador para encontrar mais paralelismo no código, a fim de minimizar as

oportunidades perdidas para desempenho. Desdobre duas iterações do loop em seu código reordenado do Exercício 3.5.

c. [10] <3.1, 3.2> Que ganho de velocidade você obteve? (Neste exercício, basta colorir as instruções da iteração $N + 1$ de verde para distingui-las das instruções da iteração N; se você estivesse realmente desdobrando o loop, teria de reatribuir os registradores para impedir colisões entre as iterações.)

3.7 [15] <3.1> Os computadores gastam a maior parte do tempo nos loops, de modo que as iterações de loop são ótimos locais para encontrar especulativamente mais trabalho para manter os recursos da CPU ocupados. Porém, nada é tão fácil; o compilador emitiu apenas uma cópia do código desse loop, de modo que, embora múltiplas iterações estejam tratando dados distintos, elas parecerão usar os mesmos registradores. Para evitar a colisão de uso de registrador por múltiplas iterações, renomeamos seus registradores. A Figura 3.49 mostra o código de exemplo que gostaríamos que nosso hardware renomeasse. Um compilador poderia ter simplesmente desdobrado o loop e usado registradores diferentes para evitar conflitos, mas, se esperarmos que nosso hardware desdobre o loop, ele também terá de fazer a renomeação de registrador. Como? Considere que seu hardware tenha um pool de registradores temporários (vamos chamá-los de registradores T e considerar que existam 64 deles, de T0 a T63) que ele pode substituir por registradores designados pelo compilador. Esse hardware de renomeação é indexado pela designação do registrador de origem, e o valor na tabela é o registrador T do último destino que designou esse registrador. (Pense nesses valores de tabela como produtores e nos registradores de origem como consumidores; não importa muito onde o produtor coloca seu resultado, desde que seus consumidores possam encontrá-lo.) Considere a sequência de código na Figura 3.49. Toda vez que você encontrar um registrador de destino no código, substitua o próximo T disponível, começando com T9. Depois atualize todos os registradores de origem adequadamente, de modo que as dependências de dados verdadeiras sejam mantidas. Mostre o código resultante. (*Dica:* Ver a Figura 3.50).

3.8 [20] <3.4> O Exercício 3.7 explorou a renomeação simples de registradores: quando o renomeador de registrador do hardware vê um registrador de origem, substitui o registrador T de destino da última instrução a ter designado esse registrador de origem. Quando a tabela de renomeação encontra um

```
Loop:    LD        F4,0(Rx)
I0:      MULTD     F2,F0,F2
I1:      DIVD      F8,F4,F2
I2:      LD        F4,0(Ry)
I3:      ADDD      F6,F0,F4
I4:      SUBD      F8,F8,F6
I5:      SD        F8,0(Ry)
```

FIGURA 3.49 Exemplo de código para prática de renomeação de registrador.

```
I0:      LD       T9,0(Rx)
I1:      MULTD    T10,F0,T9
...
```

FIGURA 3.50 *Dica:* Saída esperada do renomeamento de registrador.

registrador de destino, ela o substitui pelo próximo T disponível. Mas os projetos superescalares precisam lidar com múltiplas instruções por ciclo de clock em cada estágio na máquina, incluindo a renomeação de registrador. Um processador escalar simples, portanto, pesquisaria os mapeamentos de registrador de origem para cada instrução e alocaria um novo mapeamento de destino por ciclo de clock. Os processadores superescalares precisam ser capazes de fazer isso também, mas teriam de garantir que quaisquer relacionamentos destino para origem entre as duas instruções concorrentes fossem tratados corretamente. Considere a sequência de código de exemplo na Figura 3.51 e que gostaríamos de renomear simultaneamente as duas primeiras instruções. Considere ainda que os próximos dois registradores T disponíveis a serem usados sejam conhecidos no início do ciclo de clock em que essas duas instruções estão sendo renomeadas. Conceitualmente, o que queremos é que a primeira instrução faça suas pesquisas na tabela de renomeação e depois atualize a tabela por seu registrador T de destino. Depois, a segunda instrução faria exatamente a mesma coisa e, portanto, qualquer dependência entre instruções seria tratada corretamente. Mas não existe tempo suficiente para escrever essa designação de registrador T na tabela de renomeação e depois pesquisá-la novamente para a segunda instrução, tudo no mesmo ciclo de clock. Em vez disso, essa substituição de registrador precisa ser feita ao vivo (em paralelo com a atualização da tabela de renomeação de registrador). A Figura 3.52 mostra um diagrama de circuito usando multiplexadores e comparadores que conseguirá fazer a renomeação de registrador necessária no ato. Sua tarefa é mostrar o estado ciclo por ciclo da tabela de renomeação para cada instrução do código mostrado na Figura 3.51. Considere que a tabela começa com cada entrada igual ao seu índice (T0 = 0; T1 = 1, ...).

3.9 [5] <3.4> Se você já se confundiu com relação ao que um renomeador de registrador precisa fazer, volte ao código assembly que está executando e pergunte a si mesmo o que deve acontecer para que o resultado correto seja obtido. Por exemplo, considere uma máquina superescalar de três vias renomeando estas três instruções simultaneamente:

```
ADDI R1, R1, R1
ADDI R1, R1, R1
ADDI R1, R1, R1
```

3.10 [20] <3.4, 3.9> Projetistas de palavras de instrução muito longas (VLIW) têm algumas escolhas básicas a fazer com relação a regras de arquitetura para uso de registrador. Suponha que um VLIW seja projetado com pipelines de execução com autodrenagem: quando uma operação for iniciada, seus resultados aparecerão no registrador de destino no máximo *L* ciclos mais tarde (onde *L* é a latência da operação). Nunca existem registradores suficientes, de modo que há uma tentativa de espremer o uso máximo de registradores que existem. Considere a Figura 3.53. Se os loads tiverem uma latência de 1 + 2 ciclos,

```
I0:     SUBD     F1,F2,F3
I1:     ADDD     F4,F1,F2
I2:     MULTD    F6,F4,F1
I3:     DIVD     F0,F2,F6
```

FIGURA 3.51 Exemplo de código para renomeação de registrador superescalar.

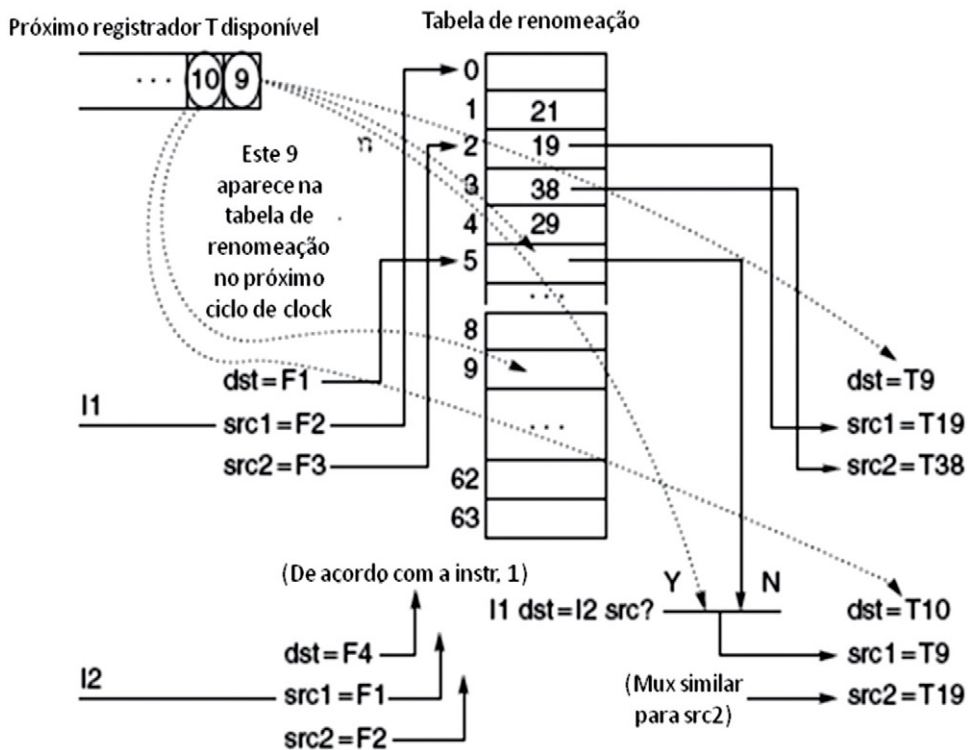



FIGURA 3.52 Tabela de renomeação e lógica de substituição em ação para máquinas superescalares. (Observe que src é a fonte e dest é o destino.)

```
Loop: LW      R4,0(R0) ;  ADDI     R11,R3,#1
      LW      R5,8(R1) ;  ADDI     R20,R0,#1
      <stall>
      ADDI    R10,R4,#1;
      SW      R7,0(R6);   SW       R9,8(R8)
      ADDI    R2,R2,#8
      SUB     R4,R3,R2
      BNZ     R4,Loop
```

FIGURA 3.53 Código VLIW de exemplo com dois adds, dois loads e dois stalls.

desdobre esse loop uma vez e mostre como um VLIW capaz de dois loads e dois adds por ciclo pode usar o número mínimo de registradores, na ausência de quaisquer interrupções ou stalls no pipeline. Dê exemplo de um evento que, na presença de pipelines de autodrenagem, possa romper essa canalização e gerar resultados errados.

3.11 3.11 [10/10/10] <3.3> Considere uma microarquitetura de único pipeline em cinco estágios (load, decodificação, execução, memória, escrita) e o código na Figura 3.54. Todas as operações são de um ciclo, exceto LW e SW, que são de 1 + 2 ciclos, e os desvios são de 1 + 1 ciclo. Não existe adiantamento. Mostre as fases de cada instrução por ciclo de clock para uma iteração do loop.

a. [10] <3.3> Quantos ciclos de clock por iteração do loop são perdidos para o overhead de desvio?

```
Loop:   LW      R3,0(R0)
        LW      R1,0(R3)
        ADDI    R1,R1,#1
        SUB     R4,R3,R2
        SW      R1,0(R3)
        BNZ     R4, Loop
```

FIGURA 3.54 Código de loop para o Exercício 3.11.

b. [10] <3.3> Considere um previsor de desvio estático capaz de reconhecer um desvio ao contrário no estágio de decodificação. Quantos ciclos de clock são desperdiçados no overhead de desvio?

c. [10] <3.3> Considere um previsor de desvio dinâmico. Quantos ciclos são perdidos em uma previsão correta?

3.12 [15/20/20/10/20] <3.4, 3.7, 3.14> Vamos considerar o que o escalonamento dinâmico poderia conseguir aqui. Considere uma microarquitetura como a da Figura 3.55. Suponha que as ALUs possam fazer todas as operações aritméticas (MULTD, DIVD, ADDD, ADDI, SUB) e desvios, e que a estação de reserva (RS) possa enviar no máximo uma operação para cada unidade funcional por ciclo (uma operação para cada ALU mais uma operação de memória para a unidade de LD/ST).

a. [15] <3.4> Suponha que todas as instruções da sequência na Figura 3.48 estejam presentes no RS, sem que qualquer renomeação precise ser feita. Destaque quaisquer instruções no código onde a renomeação de registrador melhoraria o desempenho. (*Dica:* Procure hazards RAW e WAW. Considere as mesmas latências de unidade funcional da Figura 3.48.)

b. [20] <3.4> Suponha que a versão com registrador renomeado do código do item *a* esteja residente na RS no ciclo de clock *N*, com latências conforme indicado na Figura 3.48. Mostre como a RS deverá enviar essas instruções fora de ordem, clock por clock, para obter o desempenho ideal nesse código. (Considere as mesmas restrições de RS do item *a*. Considere também que os resultados precisam ser escritos na RS antes que estejam disponíveis para uso, ou seja, sem bypassing.) Quantos ciclos de clock a sequência de código utiliza?

c. [20] <3.4> O item *b* permite que a RS tente escalonar essas instruções de forma ideal. Mas, na realidade, a sequência de instruções inteira — em que

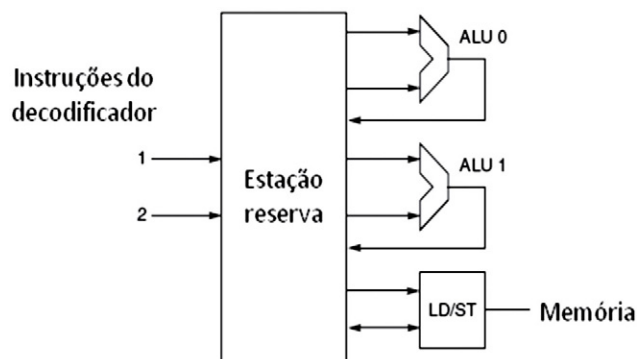



FIGURA 3.55 Microarquitetura fora de ordem.

estamos interessados — normalmente não está presente na RS. Em vez disso, diversos eventos apagam a RS e, quando novos fluxos de sequência de código entram no decodificador, a RS precisa enviar o que ela tem. Suponha que a RS esteja vazia. No ciclo 0, as duas primeiras instruções dessa sequência com registrador renomeado aparecem na RS. Considere que é necessário um ciclo de clock para enviar qualquer operação e que as latências da unidade funcional sejam como apareceram no Exercício 3.2. Considere ainda que o front-end (decodificador/renomeador de registrador) continuará a fornecer duas novas instruções por ciclo de clock. Mostre a ordem, ciclo por ciclo, de despacho da RS. Quantos ciclos de clock essa sequência de código exige agora?

d. [10] <3.14> Se você quisesse melhorar os resultados do item *c*, quais teriam ajudado mais: 1) outra ALU; 2) outra unidade de LD/ST; 3) bypassing total de resultados da ALU para operações subsequentes; 4) cortar a latência mais longa ao meio? Qual é o ganho de velocidade?

e. [20] <3.7> Agora vamos considerar a especulação, o ato de apanhar, decodificar e executar além de um ou mais desvios condicionais. Nossa motivação para fazer isso é dupla: o escalonamento de despacho que vimos no item *c* tinha muitas nops, e sabemos que os computadores gastam a maior parte do seu tempo executando loops (implicando que o desvio de volta ao topo do loop é bastante previsível). Os loops nos dizem onde encontrar mais trabalho a fazer; nosso escalonamento de despacho escasso sugere que temos oportunidades para fazer algum trabalho mais cedo do que antes. No item *d*, você descobriu o caminho crítico através do loop. Imagine gerar uma segunda cópia desse caminho no escalonamento que você obteve no item *b*. Quantos ciclos de clock a mais seriam necessários para realizar o trabalho de dois loops (supondo que todas as instruções estejam residentes na RS)? (Considere que todas as unidades funcionais sejam totalmente canalizadas.)

Exercícios

3.13 [25] <3.13> Neste exercício, você vai explorar os trade-offs de desempenho entre três processadores que empregam diferentes tipos de multithreading. Cada um desses processadores é superescalar, o uso pipelines em ordem requer um stall fixo de três ciclos seguindo todos os loads e desvios, e tem caches L1 idênticas. Instruções do mesmo thread enviados no mesmo ciclo são lidas na ordem do programa e não devem conter quaisquer dependências de dados ou controle.

- O processador A é uma arquitetura superescalar SMT capaz de enviar até duas instruções por ciclo de dois threads.
- O processador B é uma arquitetura MT fina capaz de enviar até quatro instruções por ciclo de um único thread e muda de thread a qualquer stall de pipeline.
- O processador C é uma arquitetura MT grossa capaz de enviar até oito instruções por ciclo de um thread único e muda de thread a cada falha de cache L1.

Nossa aplicação é um buscador de lista que verifica uma região de memória à procura de um valor específico em R9, na faixa de endereços especificadas em R16 e R17. Ela é paralelizada dividindo o espaço de busca em quatro blocos contíguos de tamanho igual e designando um thread de busca para cada bloco (gerando

quatro threads). A maior parte do runtime de cada thread é gasto no seguinte corpo de loop:

```
loop:  LD R1,0(R16)
       LD R2,8(R16)
       LD R3,16(R16)
       LD R4,24(R16)
       LD R5,32(R16)
       LD R6,40(R16)
       LD R7,48(R16)
       LD R8,56(R16)
       BEQAL R9,R1,match0
       BEQAL R9,R2,match1
       BEQAL R9,R3,match2
       BEQAL R9,R4,match3
       BEQAL R9,R5,match4
       BEQAL R9,R6,match5
       BEQAL R9,R7,match6
       BEQAL R9,R8,match7
       DADDIU R16,R16,#64
       BLT R16,R17,loop
```

Suponha o seguinte:

- É usada uma barreira para garantir que todos os threads comecem simultaneamente.
- A primeira falha de cache L1 ocorre depois de duas iterações do loop.
- Nenhum dos desvios BEQAL é tomado.
- O BLT é sempre tomado.
- Todos os três processadores escalonam threads de modo round-robin.

Determine quantos ciclos são necessários para cada processador completar as duas primeiras iterações do loop.

3.14 [25/25/25] <3.2, 3.7> Neste exercício, examinamos como técnicas de software podem extrair paralelismo de nível de instrução (ILP) em um loop comum vetorial. O loop a seguir é o chamado loop DAXPY (*aX* mais *Y* de precisão dupla) e é a operação central na eliminação gaussiana. O código a seguir implementa a operação DAPXY, $Y = aX + Y$, para um vetor de comprimento 100. Inicialmente, R1 é configurado para o endereço de base do array *X* e R2 e configurado para o endereço de base de *Y*:

```
DADDIU  R4,R1,#800  ; R1 = limite superior para X
foo:    L.D      F2,0(R1)    ; (F2) = X(i)
        MUL.D    F4,F2,F0    ; (F4) = a*X(i)
        L.D      F6,0(R2)    ; (F6) = Y(i)
        ADD.D    F6,F4,F6    ; (F6) = a*X(i) + Y(i)
        S.D      F6,0(R2)    ; Y(i) = a*X(i) + Y(i)
        DADDIU   R1,R1,#8    ; incrementa índice Y
        DADDIU   R2,R2,#8    ; incrementa índice Y
        DSLTU    R3,R1,R4    ; teste: continua o loop?
        BNEZ     R3,foo      ; loop se necessário
```

Considere as latências de unidade funcional mostradas na tabela a seguir. Considere também um desvio atrasado de um ciclo que se resolve no estágio ID e que os resultados são totalmente contornados.

Instrução produzindo o resultado	Instrução usando o resultado	Latência em ciclos de clock
Multiplicação de PF	Op ALU PF	6
Soma de PF	Op ALU PF	4
Multiplicação de PF	Store de PF	5
Soma de PF	Store de PF	4
Operações com inteiros e todos os loads	Any	2

a. [25] <3.2> Considere um pipeline de despacho único. Mostre como seria o loop não escalonado pelo compilador e depois escalonado pelo compilador, tanto para operação de ponto flutuante como para atrasos de desvio, incluindo quaisquer stalls ou ciclos de clock ociosos. Qual é o tempo de execução (em ciclos) por elemento do vetor resultante, *Y*, não escalonado e escalonado? Quão mais rápido o clock deveria ser para que o hardware do processador pudesse igualar sozinho a melhoria de desempenho atingida pelo compilador de escalonamento? (Ignore possíveis efeitos da maior velocidade de clock sobre o desempenho do sistema.)

b. [25] <3.2> Considere um pipeline de despacho único. Expanda o loop quantas vezes forem necessárias para escaloná-lo sem nenhum stall, ocultando as instruções de overhead do loop. Quantas vezes o loop deve ser expandido? Mostre o escalonamento de instruções. Qual é o tempo de execução por elemento do resultado?

c. [25] <3.7> Considere um processador VLIW com instruções que contêm cinco operações, como mostrado na Figura 3.16. Vamos comparar dois graus de expansão de loop. Primeiro, expanda o loop seis vezes para extrair ILP e escaloná-lo sem nenhum stall (ou seja, ciclos de despacho completamente vazios), ocultando as instruções de overhead de loop. Então, repita o processo, mas expanda o loop 10 vezes. Ignore o slot de atraso de desvio. Mostre os dois escalonamentos. Qual é o tempo de instrução, por elemento, do vetor resultado para cada escalonamento? Que porcentagem dos slots de operação é usada em cada escalonamento? Em quanto o tamanho do código difere nos dois escalonamentos? Qual é a demanda total do registrador para esses escalonamentos?

3.15 [20/20] <3.4, 3.5, 3.7, 3.8> Neste exercício, vamos examinar como variações no algoritmo de Tomasulo se comportam quando executam o loop do Exercício 3.14. As unidades funcionais (FUs) são descritas na tabela a seguir.

Tipo de FU	Ciclos em EX	Número de FUs	Número de estações de reserva
Inteiro	1	1	5
Somador de PF	10	1	3
Multiplicador de PF	15	1	2

Considere o seguinte:

- As unidades funcionais não são pipelined.
- Não há adiandamento entre as unidades funcionais; os resultados são comunicados pelo barramento comum de dados (CDB).

- O estágio de execução (EX) realiza o cálculo efetivo de endereço e os acessos à memória para loads e stores. Assim, o pipeline é IF/ID/IS/EX/WB.
- Loads requerem um ciclo de clock.
- Os estágios de resultados de despacho (IS) e write-back (WB) requerem um ciclo de clock cada um.
- Há cinco slots de buffer de carregamento e cinco slots de buffer de armazenamento.
- Considere que a instrução Branch on Not Equal do Zero (BNEZ) requer um ciclo de clock.

a. [20] <3.4, 3.5> Para este problema, use o pipeline MIPS de Tomasulo de despacho único da Figura 3.6 com as latências de pipeline da tabela anterior. Mostre o número de ciclos de stall para cada instrução e em que ciclo de clock cada uma delas começa a ser executada (ou seja, entra no seu primeiro ciclo EX) para três iterações do loop. Quantos ciclos cada iteração de loop leva? Dê sua resposta em forma de tabela com os seguintes títulos de coluna:
- Iteração (número da iteração do loop)
- Instrução
- Envia (ciclo em que a instrução é enviada)
- Executa (ciclo em que a instrução é executada)
- Acesso à memória (ciclo em que a memória é acessada)
- CDB de gravação (ciclo em que o resultado é gravado no CDB)
- Comentário (descrição de qualquer evento que a instrução esteja aguardando)

Mostre três iterações do loop na sua tabela. Você pode ignorar a primeira instrução.

b. [20] <3.7, 3.8> Repita o procedimento do item *a*, mas desta vez considere um algoritmo de Tomasulo de dois despachos e uma unidade de ponto flutuante totalmente pipelined (FPU).

3.16 [10] <3.4> O algoritmo de Tomasulo apresenta uma desvantagem: somente um resultado pode ser computado por clock por CDB. Use a configuração de hardware e latências da questão anterior e encontre uma sequência de código de não mais de 10 instruções onde o algoritmo de Tomasulo deve sofrer stall, devido à contenção de CDB. Indique onde isso ocorre na sua sequência.

3.17 [20] <3.3> Um previsor de desvio de correlação (m,n) usa o comportamento dos m desvios executados mais recentemente para escolher entre 2^m previsores, cada qual previsor de n bits. Um previsor local de dois níveis funciona de modo similar, mas só rastreia o comportamento passado de cada desvio individual para prever o comportamento futuro.

Existe um trade-off de projeto envolvido com tais previsores. Previsores de correlação requerem pouca memória para histórico, o que permite a eles manter previsores de 2 bits para um grande número de desvios individuais (reduzindo a probabilidade das instruções de desvio reutilizarem o mesmo previsor), enquanto previsores locais requerem substancialmente mais memória para manter um histórico e, assim, são limitados a rastrear um número relativamente menor de instruções de desvio. Neste exercício, considere um previsor de correlação (1,2) que pode rastrear quatro desvios (requerendo 16 bits) em comparação com um previsor local (1.2) que pode rastrear dois desvios usando a mesma quantidade de memória. Para os resultados de desvio a seguir, forneça cada previsão, a entrada de tabela usada para realizar a previsão, quaisquer atualizações na tabela como resultado da previsão e a taxa final de previsões incorretas de cada previsor.

Suponha que todos os desvios até este ponto tenham sido tomados. Inicialize cada previsor com o seguinte:

Previsor de correlação			
Entrada	**Desvio**	**Último resultado**	**Previsão**
0	0	T	T com uma previsão incorreta
1	0	NT	NT
2	1	T	NT
3	1	NT	T
4	2	T	T
5	2	NT	T
6	3	T	NT com uma previsão incorreta
7	3	NT	NT

Previsor local			
Entrada	**Desvio**	**Últimos dois resultados (o da direita é o mais recente)**	**Previsão**
0	0	T,T	T com uma previsão incorreta
1	0	T,NT	NT
2	0	NT,T	NT
3	0	NT	T
4	1	T,T	T
5	1	T,NT	T com uma previsão incorreta
6	1	NT,T	NT
7	1	NT,NT	NT

Desvio PC (endereço de palavra)	Resultado
454	T
543	NT
777	NT
543	NT
777	NT
454	T
777	NT
454	T
543	T

3.18 [10] <3.9> Considere um processador altamente pipelined para o qual tenhamos implementado um buffer de alvos de desvio somente para os desvios condicionais. Considere que a penalidade de previsão incorreta é sempre de cinco ciclos e a penalidade de falha de buffer é sempre de três ciclos. Considere uma taxa de acerto de 90%, precisão de 90% e frequência de desvio de 15%. Quão mais rápido é o processador com o buffer de alvo de desvio comparado a um

processador que tenha uma penalidade de desvio fixa de dois ciclos? Considere um ciclo-base de clock por instrução (CPI) sem stalls de desvio de um.

3.19 [10/5] <3.9> Considere um buffer de alvos de desvio que tenha penalidades de zero, dois e dois ciclos de clock para previsão correta de desvio condicional, previsão incorreta e uma falha de buffer, respectivamente. Considere também um projeto de buffer de alvo de desvio que distingue desvios condicionais e não condicionais, armazenando os endereços de alvo para um desvio condicional e a instrução-alvo para um desvio não condicional.

a. [10] <3.9> Qual é a penalidade em ciclos de clock quando um desvio não condicional é encontrado no buffer?

b. [10] <3.9> Determine a melhoria da dobra de desvios para desvios não condicionais. Suponha uma taxa de acerto de 90%, uma frequência de desvio não condicional de 5% e uma penalidade de dois ciclos para uma falha de buffer. Quanta melhoria é obtida por essa modificação? Quão alta deve ser a taxa de acerto para essa melhoria gerar um ganho de desempenho?

CAPÍTULO 4

Paralelismo em nível de dados em arquiteturas vetoriais, SIMD e GPU[1]

Chamamos esses algoritmos de *paralelismo de dados* porque seu paralelismo vem de operações simultâneas através de grandes conjuntos de dados, em vez de múltiplos threads de controle.

W. Daniel Hillis e Guy L. Steele
"Data Parallel Algorithms", *Comm. ACM* (1986).

Se você estivesse arando um campo, o que preferiria usar: dois bois fortes ou 1.024 galinhas?

Seymour Cray, Pai do Supercomputador,
(*defendendo dois poderosos processadores vetoriais em vez de vários processadores simples*).

4.1 INTRODUÇÃO

Uma questão para a arquitetura de simples instrução e múltiplos dados (SIMD), apresentada no Capítulo 1, é de que a largura do conjunto de aplicações tem paralelismo significativo em nível de dados (DLP). Cinquenta anos depois, a resposta não são só os cálculos orientados para a matriz da computação científica, mas também para o processamento de imagens e sons orientado para a mídia. Além disso, como uma única instrução pode lançar muitas operações de dados, o SIMD é potencialmente mais eficiente em termos de energia do que múltiplas instruções e múltiplos dados (MIMD), que precisam buscar e executar uma instrução por operação de dados. Essas duas respostas tornam o SIMD atraente para dispositivos pessoais móveis. Por fim, talvez a maior vantagem do SIMD em comparação ao MIMD seja que o programador continua a pensar sequencialmente e, ainda assim, atinge um ganho de velocidade ao realizar operações de dados paralelas.

Este capítulo abrange três variações do SIMD: arquiteturas vetoriais, extensões de conjunto de instrução SIMD para multimídia e unidades de processamento gráfico (GPUs).[1]

A primeira variação, que antecede as outras duas em mais de 30 anos, significa essencialmente a execução em pipeline de muitas operações de dados. Essas *arquiteturas vetoriais* são mais fáceis de entender e compilar do que outras variações de SIMD, mas até bem recentemente eram consideradas muito caras para os microprocessadores. Parte desse custo era referente a transistores e parte à largura de banda suficiente para a DRAM, dada a dependência generalizada das caches para atender às demandas de desempenho de memória em microprocessadores convencionais.

A segunda variação SIMD pega esse nome emprestado para representar operações simultâneas de dados paralelos (Simultaneous Parallel Data Operations) e é encontrada na maioria das arquiteturas de conjunto de instruções atuais que suportam aplicações multimídia. Para arquiteturas ×86, as extensões de instruções SIMD começaram com o MMX (extensões multimídia) em 1996, seguidas por diversas versões SSE (extensões SIMD para streaming) na década seguinte e continuam com as AVX (extensões vetoriais avançadas). Muitas vezes, para obter a maior taxa de computação de um computador ×86, você precisa usar essas instruções SIMD, especialmente para programas de ponto flutuante.

A terceira variação do SIMD vem da comunidade GPU, oferecendo maior desempenho potencial do que o encontrado nos computadores multicore tradicionais de hoje. Embora as GPUs compartilhem características com as arquiteturas vetoriais, elas têm suas próprias características, em parte devido ao ecossistema no qual evoluíram. Esse ambiente tem um sistema de processador e um sistema de memória, além da GPU e de sua memória gráfica. De fato, para reconhecer essas distinções, a comunidade GPU se refere a esse tipo de arquitetura como *heterogênea*.

Por problemas com muito paralelismo de dados, as três variações de SIMD compartilham a vantagem de serem mais fáceis para os programadores do que a clássica programação MIMD. Para colocar em perspectiva a importância do SIMD *versus* o MIMD, a Figura 4.1 plota o número de núcleos para o MIMD *versus* o número de operações de 32 bits e 64 bits por ciclo de clock no modo SIMD para computadores ×86 ao longo do tempo.

Para os computadores ×86, esperamos ver dois núcleos adicionais por chip a cada dois anos e a largura SIMD dobrar a cada quatro anos. Dadas essas suposições, ao longo da próxima década, o ganho potencial de velocidade do paralelismo SIMD será duas vezes o do paralelismo MIMD. Portanto, é igualmente importante entender o paralelismo SIMD como paralelismo MIMD, embora recentemente o último tenha recebido muito mais destaque. Para aplicações com paralelismo em nível de dados e paralelismo em nível de thread, o ganho potencial de velocidade em 2020 terá magnitude maior do que hoje.

O objetivo deste capítulo é fazer que os arquitetos entendam por que os vetores são mais gerais do que o SIMD de multimídia, assim como as similaridades e diferenças entre as arquiteturas vetoriais e as de GPU. Como as arquiteturas vetoriais são superconjuntos das instruções SIMD multimídia, incluindo um modelo melhor para compilação, e as GPUs compartilham diversas similaridades com as arquiteturas vetoriais, começamos com arquiteturas vetoriais para estabelecer a base para as duas seções a seguir. A seção seguinte apresenta as arquiteturas vetoriais, e o Apêndice G vai muito mais fundo no assunto.

---

[1] Este capítulo se baseia em material do Apêndice F, "Processadores Vetoriais", de Krste Asanovic, e do Apêndice G, "Hardware e Software para VLIW e EPIC" da 4ª edição deste livro; em material do Apêndice A, "Graphics and Computing GPUs", de John Nickolls e David Kirk, da 4ª edição de *Computer Organization and Design*; e, em menor escala, em material de "Embracing and Extending 20th-Century Instruction Set Architectures", de Joe Gebis e David Patterson, *IEEE Computer*, abril de 2007.

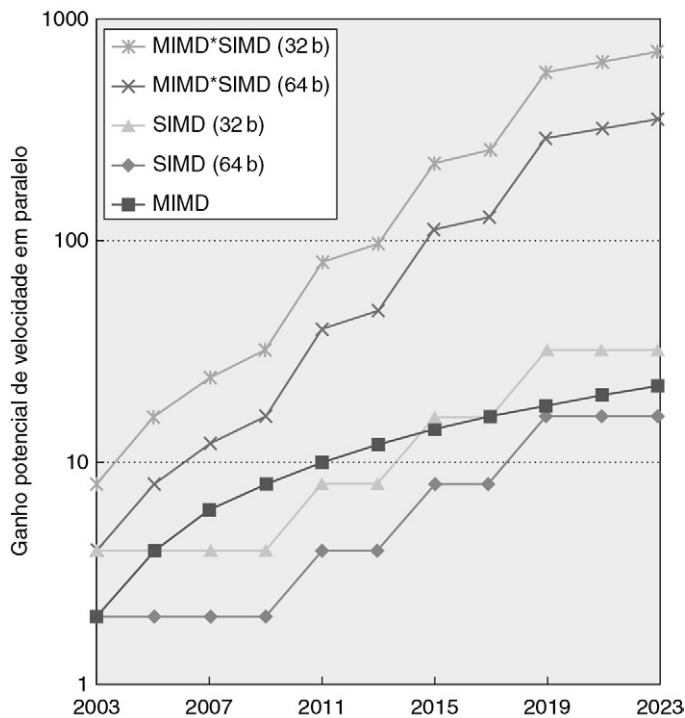



FIGURA 4.1 Ganho de vista potencial através de paralelismo de MIMD, SIMD e tanto MIMD quanto SIMD ao longo do tempo para computadores ×86.
Esta figura supõe que dois núcleos por chip para MIMD serão adicionados a cada dois anos e o número de operações para SIMD vai dobrar a cada quatro anos.

4.2 ARQUITETURA VETORIAL

> O modo mais eficiente de executar uma aplicação vetorizável é um processador vetorial.
>
> **Jim Smith**
> *International Symposium on Computer Architecture* (1994)

As arquiteturas vetoriais coletam conjuntos de elementos de dados espalhados pela memória, os colocam em arquivos de registradores sequenciais, operam sobre dados nesses arquivos de registradores e então dispersam os resultados de volta para a memória. Uma única instrução opera sobre vetores de dados, que resulta em dúzias de operações registrador-registrador em elementos de dados independentes.

Esses grandes arquivos de registradores agem como buffers controlados pelo compilador, tanto para ocultar a latência de memória quanto para aproveitar a largura de banda da memória. Como carregamentos e armazenamentos vetorias são fortemente pipelined, o programa paga pela grande latência de memória somente uma vez por carregamento ou armazenamento vetorial em comparação a uma vez por elemento, amortizando assim a latência ao longo de cerca de 64 elementos. De fato, os programas vetoriais lutam para manter a memória ocupada.

VMIPS

Começamos com um processador vetorial que consiste nos principais componentes mostrados na Figura 4.2. Esse processador, que é livremente baseado no Cray-1, é o alicerce para a discussão por quase toda esta seção. Nós o chamaremos de *VMIPS*; sua parte escalar

é MIPS e sua parte vetorial é a extensão vetorial lógica do MIPS. O restante desta seção examina a forma como a arquitetura básica do VMIPS está relacionada com os outros processadores.

Os principais componentes da arquitetura do conjunto de instruções do VMIPS são os seguintes:

- *Registradores vetorial.* Cada registrador vetorial é um banco de tamanho fixo mantendo um único vetor. O VMIPS possui oito registradores vetoriais, cada qual com 64 elementos. O registrador vetorial precisa fornecer portas suficientes para alimentar todas as unidades funcionais vetoriais. Essas portas vão permitir alto grau de sobreposição entre as operações vetoriais para diferentes registradores vetoriais. As portas de leitura e escrita, que totalizam pelo menos 16 portas de leitura e oito portas de escrita, estão conectadas às entradas ou saídas de unidade funcional por um par de matrizes de chaveamento crossbars.
- *Unidades funcionais vetoriais.* Cada unidade é totalmente pipelined e pode iniciar uma nova operação a cada ciclo de clock. Uma unidade de controle é necessária para detectar os riscos, sejam riscos estruturais para unidades funcionais, sejam riscos de dados em acessos de registradores. A Figura 4.2 mostra que o VMIPS possui cinco unidades funcionais. Para simplificar, focalizaremos exclusivamente as unidades funcionais de ponto flutuante.
- *Unidade carregamento-armazenamento vetorial.* Essa é uma unidade de memória vetorial que carrega ou armazena um vetor na memória. Os carregamentos e armazenamentos vetoriais do VMIPS são totalmente pipelined, de modo que as palavras podem ser movidas entre os registradores vetoriais e memória com largura de banda de uma palavra por ciclo de clock, após uma latência inicial. Normalmente, essa unidade trataria também de carregamentos e armazenamentos de escalares.
- *Um conjunto de registradores escalares.* Os registradores escalares também podem oferecer dados como entrada para as unidades funcionais vetoriais, além de calcular endereços para passar para a unidade *load/store* vetorial. Esses são os 32 registradores de uso geral normais e 32 registradores de ponto flutuante do MIPS. Uma entrada da unidade funcional vetorial trava valores escalares como lidos do banco de registradoreses escalares.

A Figura 4.3 lista as instruções vetoriais do VMIPS. No VMIPS, as operações vetoriais utilizam os mesmos nomes das operações do MIPS, mas com as letras "VV" anexadas. Assim, ADDVV.D é uma adição de dois vetores de precisão dupla. As instruções vetoriais têm como entrada um par de registradores vetoriais (ADDVV.D) ou um registrador vetorial e um registrador escalar, designado pelo acréscimo de "VS" (ADDVS.D). Neste último caso, o valor no registrador escalar será usado como entrada para todas as operações — a operação ADDVS.D acrescentará o conteúdo de um registrador escalar a cada elemento em um registrador vetorial. O valor escalar será copiado para a unidade funcional vetorial no momento da emissão. A maioria das operações vetoriais possui um registrador de destino vetorial, embora algumas (contagem de elementos) produzam um valor escalar, que é armazenado em um registrador escalar.

Os nomes LV e SV indicam load vetorial e store vetorial, e carregam ou armazenam um vetor de dados inteiros de precisão dupla. Um operando é o registrador vetorial a ser carregado ou armazenado; o outro operando, que é um registrador de uso geral do MIPS, é o endereço inicial do vetor na memória. Como veremos, além dos registradores vetoriais, precisamos de dois registradores adicionais de uso especial: os registradores de comprimento vetorial e de máscara vetorial. O primeiro é usado quando o tamanho natural do vetor não é 64, e o último é usado quando os loops envolvem declarações IF.

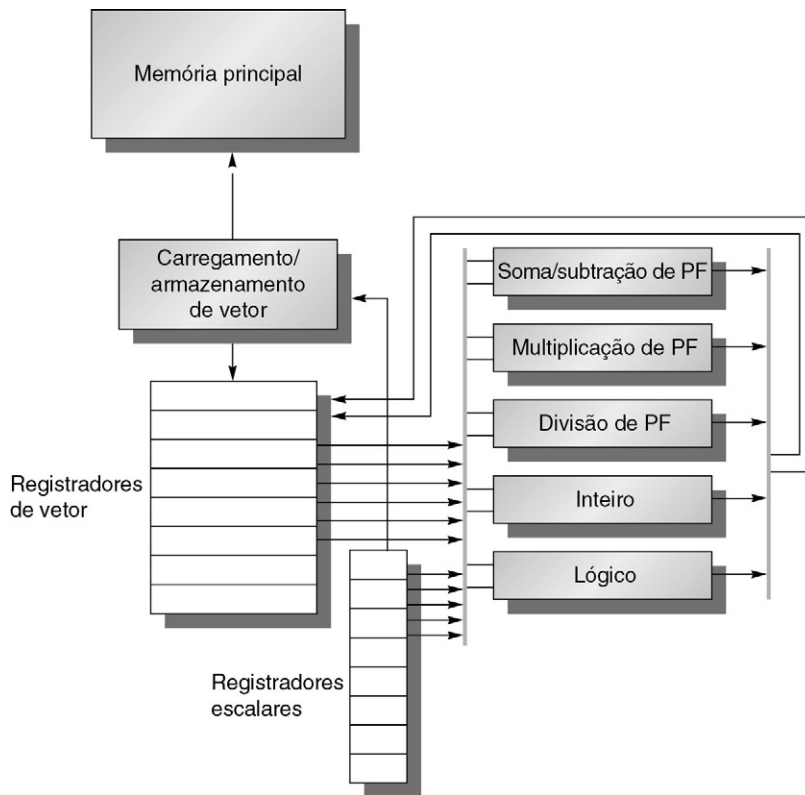



FIGURA 4.2 **Estrutura básica de uma arquitetura vetorial, VMIPS.**
Esse processador possui uma arquitetura escalar, assim como o MIPS. Há também oito registradores vetoriais de 64 elementos, e todas as unidades funcionais são unidades funcionais vetoriais. Instruções especiais vetoriais são definidas, neste capítulo, tanto para aritmética quanto para acessos à memória. A figura mostra as unidades vetoriais para operações lógicas e de inteiros, fazendo com que o VMIPS se pareça com um processador vetorial padrão, que normalmente as inclui. Porém, não vamos discutir essas unidades, exceto nos exercícios. Os registradores vetoriais e escalares têm número significativo de portas de leitura e escrita para permitir várias operações vetoriais simultâneas. Essas portas estão conectadas às entradas e saídas das unidades funcionais vetoriais por um conjunto de swichtes crossbars (mostrados em linhas cinza grossas) conectadas a essas portas de entradas e saídas das unidades funcionais vetoriais.

A barreira da potência levou os arquitetos a avaliarem arquiteturas que possam apresentar alto desempenho sem os custos de energia e a complexidade de processadores superescalares com processamento fora de ordem. As instruções vetoriais são um parceiro natural para essa tendência, já que os arquitetos podem usá-las para aumentar o desempenho de simples processadores escalares em ordem sem aumentar muito as demandas de energia e complexidade de projeto. Na prática, os desenvolvedores podem expressar muitos dos programas que funcionavam bem em projetos complexos fora de ordem com mais eficiência como paralelismo em nível de dados na forma de instruções vetoriais, tal como mostrado por Kozyrakis e Patterson (2002).

Com uma instrução vetorial, o sistema pode realizar as operações sobre os elementos de dados do vetor de muitas maneiras, incluindo como operar em muitos elementos simultaneamente. Essa flexibilidade permite aos projetos vetoriais usar unidades de execução lentas, porém largas, sem realizar custosas verificações adicionais de dependência, como exigem os processadores superescalares.

Os vetores acomodam naturalmente tamanhos variáveis de dados. Portanto, uma interpretação de um tamanho de registrador vetorial é de 64 elementos de 64 bits, mas 128 elementos

Instrução	Operandos	Função
ADDVV.D ADDVS.D	V1,V2,V3 V1,V2,F0	Incluir elementos de V2 e V3, depois colocar cada resultado em V1. Somar F0 a cada elemento de V2, depois colocar cada resultado em V1.
SUBVV.D SUBVS.D SUBSV.D	V1,V2,V3 V1,V2,F0 V1,F0,V2	Subtrair elementos de V3 de V2, depois colocar cada resultado em V1. Subtrair F0 dos elementos de V2, depois colocar cada resultado em V1. Subtrair elementos de V2 de F0, depois colocar cada resultado em V1.
MULVV.D MULVS.D	V1,V2,V3 V1,V2,F0	Multiplicar elementos de V2 e V3, depois colocar cada resultado em V1. Multiplicar cada elemento de V2 por F0, depois colocar cada resultado em V1.
DIVVV.D DIVVS.D DIVSV.D	V1,V2,V3 V1,V2,F0 V1,F0,V2	Dividir elementos de V2 por V3, depois colocar cada resultado em V1. Dividir elementos de V2 por F0, depois colocar cada resultado em V1. Dividir F0 por elementos de V2, depois colocar cada resultado em V1.
LV	V1,R1	Carregar registrador vetorial V1 da memória começando no endereço R1.
SV	R1,V1	Armazenar registrador vetorial V1 na memória começando no endereço R1.
LVWS	V1,(R1,R2)	Carregar V1 do endereço em R1 com passo em R2, isto é, R1+ i × R2.
SVWS	(R1,R2),V1	Armazenar V1 do endereço em R1 com passo em R2, isto é, R1+ i × R2.
LVI	V1,(R1+V2)	Carregar V1 com vetor cujos elementos estão em R1+ V2(i), isto é, V2 é um índice.
SVI	(R1+V2),V1	Armazenar V1 no vetor cujos elementos estão em R1 + V2(i), isto é, V2 é um índice.
CVI	V1,R1	Criar um vetor de índice armazenando os valores 0, 1 × R1, 2 ×R1,...,63 × R1 em V1.
S--VV.D S--VS.D	V1,V2 V1,F0	Comparar os elementos (EQ, NE, GT, LT, GE, LE) em V1 e V2. Se a condição for verdadeira, colocar um 1 no vetor de bits correspondente; caso contrário, colocar 0. Colocar o vetor de bits resultante no registrador de máscara de vetor (VM). A instrução S--VS. D realiza a mesma comparação, mas usando um valor escalar como um operando.
POP	R1,VM	Contar os 1s no registrador de máscara vetorial e armazenar a contagem em R1.
CVM		Definir o registrador de máscara vetorial como todos 1s.
MTC1 MFC1	VLR,R1 R1,VLR	Mover conteúdo de R1 para o registrador de tamanho vetorial. Mover conteúdo do registrador de tamanho vetorial em R1.
MVTM MVFM	VM,F0 F0,VM	Mover conteúdo de F0 para o registrador de máscara vetorial. Mover conteúdo do registrador de máscara vetorial para F0.

FIGURA 4.3 Instruções vetoriais do VMIPS.

Apenas as operações de PF de precisão dupla aparecem. Além dos registradores vetoriais, existem dois registradores especiais, VLR (discutido na Seção F.3) e VM (discutido na Seção F.4). Esses registradores especiais são considerados como vivendo no espaço do coprocessador 1 do MIPS, junto com os registradores FPU. As operações com passo serão explicadas na Seção F.3, e os usos da criação de índice e operações carregamento-armazenamento indexadas serão explicadas mais adiante.

de 32 bits, 256 elementos de 16 bits e até mesmo 512 elementos de 8 bits são interpretações igualmente válidas. Essa multiplicidade de hardware é o motivo de uma arquitetura vetorial ser útil para aplicações multimídia e científicas.

Como os processadores vetoriais funcionam: exemplo

Um processador vetorial pode ser mais bem entendido examinando-se um loop vetorial no VMIPS. Vamos usar um problema típico vetorial, que será usado no decorrer desta seção:

$$Y = a \times X + Y$$

X e Y são vetores, inicialmente residentes na memória, e a é um escalar. Esse é o chamado loop *SAXPY* ou *DAXPY*, que forma o loop interno do benchmark Linpack. (SAXPY é a sigla para single-precision a × X plus Y; DAXPY é a sigla para double-precision a × X plus Y.)

Linpack é uma coleção de rotinas da álgebra linear, e o benchmark Limpack consite em rotinas que realizam a eliminação gaussiana. A rotina DAXPY, que implementa o loop anterior, representa uma pequena fração do código-fonte do benchmark Linpack, mas considera a maior parte do tempo de execução para esse benchmark.

Por enquanto, vamos considerar que o número de elementos, ou tamanho de um registrador vetorial (64), corresponde ao tamanho da operação vetorial em que estamos interessados (essa restrição será removida brevemente).

Exemplo Mostre o código para MIPS e VMIPS para o loop DAXPY. Considere que os endereços iniciais de *X* e *Y* estão em Rx e Ry, respectivamente.

Resposta Aqui está o código MIPS.

```
      L.D       F0,a         ;carrega escalar a
      DADDIU    R4,Rx,#512   ;último endereço a carregar
Loop: L.D       F2,0(Rx)     ;carrega X(i)
      MUL.D     F2,F2,F0     ;a × X(i)
      L.D       F4,0(Ry)     ;carrega Y(i)
      ADD.D     F4,F4,F2     ;a × X(i) + Y(i)
      S.D       F4,9(Ry),    ;armazena em Y(i)
      DADDIU    Rx,Rx,#8     ;incrementa índice em X
      DADDIU    Ry,Ry,#8     ;incrementa índice em Y
      DSUBU     R20,R4,Rx    ;calcula limite
      BNEZ      R20,Loop     ;verifica se terminou
```

Aqui está o código VMIPS para o loop DAXPY.

```
L.D       F0,a         ;carrega escalar a
LV        V1,Rx        ;carrega vetor X
MULVS.D   V2,V1,F0     ;multiplicação escalar vetorial
LV        V3,Ry        ;carrega vetor Y
ADDVV.D   V4,V2,V3     ;soma
SV        V4,Ry        ;armazena o resultado
```

A diferença mais importante é que o processador vetorial reduz bastante a largura de banda de instrução dinâmica, executando apenas seis instruções contra quase 600 para MIPS. Essa redução ocorre tanto porque as operações vetoriais trabalham sobre 64 elementos quanto porque as instruções de overhead que constituem quase metade do loop no MIPS não estão presentes no código VMIPS. Quando o compilador produz instruções vetoriais para essa sequência e o código passa grande parte do tempo sendo executado em modo vetorial, diz-se que o código está *vetorizado* ou *vetorizável*. Loops podem ser vetorizados quando não têm dependências entre as suas iterações loop, as quais são chamadas *dependências loop-carried* (Seção 4.5).

Outra diferença importante entre MIPS e VMIPS é a frequência dos interbloqueios do pipeline. No código MIPS direto, cada ADD.D precisa esperar por um MUL.D e cada S.D precisa esperar pelo ADD.D. No processador vetorial, cada instrução vetorial sofrerá stall somente para o primeiro elemento em cada vetor, e depois os elementos subsequentes fluirão suavemente pelo pipeline. Assim, stalls de pipeline são exigidos apenas uma vez por *operação* vetorial, e não uma vez por *elemento* do vetor. Os arquitetos vetoriais chamam o adiantamento de operações de elementos dependentes de *encadeamento*, onde as operações dependentes formam uma "corrente" ou "cadeia". Neste exemplo, a frequência de stall do pipeline no MIPS será cerca de 64 vezes maior do que no VMIPS. Os stalls de pipeline podem ser eliminados no MIPS usando pipelining de software ou desdobramento de loop (descrito no Apêndice H). Contudo, a grande diferença na largura de banda de instrução não pode ser reduzida.

Tempo de execução vetorial

O tempo de execução de uma sequência de operações vetoriais depende principalmente de três fatores: 1) o tamanho dos vetores do operando; 2) os riscos estruturais entre as operações; e 3) as dependências de dados. Dados o tamanho do vetor e a *taxa de iniciação*, que é a velocidade com que uma unidade vetorial consome novos operandos e produz novos resultados, podemos calcular o tempo para uma única instrução vetorial. Todos os supercomputadores modernos têm unidades funcionais vetoriais com múltiplos pipelines paralelos (ou *pistas*) que podem produzir dois ou mais resultados por ciclo de clock, mas também têm algumas unidades funcionais que não são totalmente pipelineds. Por simplicidade, nossa implementação VMIPS tem uma pista com taxa de iniciação de um elemento por ciclo de clock para operações individuais. Assim, o tempo de execução para uma única instrução vetorial é aproximadamente o tamanho do vetor.

Para simplificar a discussão sobre a execução do vetor e seu tempo, usaremos a noção de *comboio*, que é o conjunto de instruções vetoriais que podem iniciar a execução juntas em um período de clock. (Embora o conceito de comboio seja usado em compiladores vetoriais, não existe uma terminologia-padrão. Por isso, criamos o termo *comboio*.) As instruções em um comboio *não podem* conter quaisquer riscos estruturais ou de dados; se esses riscos estivessem presentes, as instruções no comboio em potencial precisariam ser seriadas e iniciadas em diferentes comboios. Para manter a análise simples, consideramos que um comboio de instruções precisa completar a execução antes que quaisquer outras instruções (escalares ou vetoriais) possam iniciar a execução.

Pode parecer que, além das sequências de instruções vetoriais com riscos estruturais, seria: as sequências de leitura com riscos de dependência leitura após gravação também deveriam estar em comboios diferentes, mas o encadeamento permite que elas estejam no mesmo comboio.

O encadeamento permite que uma operação vetorial comece assim que os elementos individuais do operando-fonte desse vetor fiquem disponíveis: os resultados da primeira unidade funcional na cadeia são "adiantados" para a segunda unidade funcional. Na prática, muitas vezes implementamos o encadeamento permitindo que o processador leia e grave um registrador vetorial particular ao mesmo tempo, embora para diferentes elementos. As primeiras implementações de encadeamento funcionavam do mesmo modo que o adiantamento em pipelines escalares, mas isso restringia a temporização dos fontes e destinos das instruções na cadeia. Implementações recentes usam o *encadeamento flexível*, que permite que uma instrução vetorial seja encadeada para qualquer outra instrução vetorial ativa, supondo que isso não gere um risco estrutural. Todas as arquiteturas vetoriais modernas suportam encadeamento flexível, que vamos considerar neste capítulo.

Para converter comboios em tempo de execução precisamos de uma medida de temporização para estimar o tempo de um comboio. Ela é chamada de *chime*, que é a unidade de tempo necessária para executar um comboio. Assim, uma sequência vetorial que consiste em m comboios é executada em m chimes, e, para um tamanho vetorial de n, isso é aproximadamente $m \times n$ ciclos de clock. Uma aproximação do chime ignora alguns overheads específicos do processador, muitos dos quais dependem do tamanho do vetor. Logo, medir o tempo em chimes é uma aproximação melhor para vetores longos. Usaremos a medida do chime em vez de ciclos de clock por resultado para indicar explicitamente que certos overheads estão sendo ignorados.

Se soubermos o número de comboios em uma sequência vetorial, saberemos o tempo de execução em chimes. Uma fonte de overhead ignorada na medição de chimes é qualquer limitação na iniciação de múltiplas instruções vetoriais em um ciclo de clock. Se apenas

uma instrução vetorial puder ser iniciada em um ciclo de clock (a realidade na maioria dos processadores vetoriais), a contagem de chime subestimará o tempo de execução real de um comboio. Como o tamanho do vetor normalmente é muito maior que o número de instruções no comboio, simplesmente consideraremos que o comboio é executado em um chime.

Exemplo Mostre como a sequência de código a seguuir é disposta em comboios, considerando uma única cópia de cada unidade funcional vetorial:

```
LV        V1,Rx      ;carrega vetor X
MULVS.D   V2,V1,F0   ;multiplicação de vetor-escalar
LV        V3,Ry      ;carrega vetor Y
ADDVV.D   V4,V2,V3   ;soma dois vetores
SV        V4,Ry      ;armazena o resultado
```

De quantos chimes essa sequência vetorial precisa? Quantos ciclos por FLOP (operação de ponto flutuante) são necessários, ignorando o overhead da emissão da instrução vetorial?

Resposta O primeiro comboio é ocupado pela primeira instrução LV. O MULVS.D depende do primeiro LV, de modo que não pode estar no mesmo comboio. A segunda instrução LV pode estar no mesmo comboio de MULVS.D. O ADDV.D é dependente do segundo LV, de modo que precisa vir em um terceiro comboio, e por fim o SV depende do ADDVV.D, de modo que precisa vir em um comboio seguinte. Isso leva ao seguinte leiaute de instruções vetoriais nos comboios:

```
1. LV     MULVS.D
2. LV     ADDVV.D
3. SV
```

A sequência exige três comboios. Como a sequência usa um total de três chimes e existem duas operações de ponto flutuante por resultado, o número de ciclos por FLOP é 1,5 (ignorando qualquer overhead de emissão de instrução vetorial). Observe que, embora permitíssemos que MULVS.D e LV fossem executadas no primeiro comboio, a maioria das máquinas vetoriais usará dois ciclos de clock para iniciar as instruções.

Este exemplo mostra que a aproximação chime é razoavelmente precisa para vetores longos. Por exemplo, para vetores de 64 elementos, o tempo em chimes é 3, então a sequência levaria cerca de 64×3 ou 192 ciclos de clock. O overhead de despachar comboios em dois ciclos de clock separados seria pequeno.

Outra fonte de overhead é muito mais significativa do que a limitação de despacho. A fonte mais importante de overhead, ignorada pelo modelo de chime, é o *tempo de início* do vetor. O tempo de início vem da latência de pipelining da operação vetorial e é determinado principalmente pela profundidade do pipeline para a unidade funcional utilizada. Para VMIPS, vamos usar as mesmas profundidades de pipeline do Cray-1, embora as latências em processadores mais modernos tenham aumentado, especialmente para carregamentos vetoriais. Todas as unidades funcionais são totalmente pipelined. As profundidades de pipeline são de seis ciclos de clock por soma de ponto flutuante, sete para multiplicação de ponto flutuante, 20 para divisão de ponto flutuante e 12 para carregamento vetorial.

Dados esses conceitos básicos vetoriais, as próximas subseções vão dar otimizações que melhoram o desempenho ou diminuem os tipos de programas que podem ser bem executados em arquiteturas vetoriais. Em particular, elas vão responder às questões:

- Como um processador vetorial executa um único vetor mais rápido do que um elemento por ciclo de clock? Múltiplos elementos por ciclo de clock melhoram o desempenho.
- Como um processador vetorial trata programas em que os comprimentos dos vetores não são iguais ao comprimento do registrador vetorial (64 para VMIPS)? Como a maioria dos vetores de aplicação não corresponde ao comprimento de vetor da arquitetura, precisamos de uma solução eficiente para esse caso comum.
- O que acontece quando existe uma declaração IF dentro do código a ser vetorizado? Mais código pode ser vetorizado se pudermos lidar eficientemente com declarações condicionais.
- O que um processador vetorial precisa do sistema de memória? Sem largura de banda de memória suficiente, a execução vetorial pode ser fútil.
- Como um processador vetorial lida com matrizes multidimensionais? Essa popular estrutura de dados deve ser vetorizada para arquiteturas vetoriais para funcionar bem.
- Como um processador vetorial lida com matrizes dispersas? Essa popular estrutura de dados também deve ser vetorizada.
- Como você programa um computador vetorial? Inovações arquiteturais que não correspondam à tecnologia de compilador podem não ser amplamente utilizadas.

O restante desta seção apresentará cada uma dessas otimizações da arquitetura vetorial, e o Apêndice G mostrará mais detalhes.

Múltiplas pistas: além de um elemento por ciclo de clock

Uma das maiores vantagens de um conjunto de instruções vetoriais é que ele permite que o software passe uma grande quantidade de trabalho paralelo para o hardware usando uma única instrução curta. Uma única instrução vetorial pode incluir entre dezenas e centenas de operações independentes, porém ser codificada com o mesmo número de bits de uma instrução escalar convencional. A semântica paralela de uma instrução vetorial permite que uma implementação execute essas operações elementares usando uma unidade funcional de pipeline profundo, como na implementação VMIPS que estudamos até aqui, ou usando um array de unidades funcionais paralelas ou uma combinação de unidades funcionais paralelas e em pipeline. A Figura 4.4 ilustra como o desempenho do vetor pode ser melhorado usando pipelines paralelos para executar uma instrução de adição vetorial.

O conjunto de instruções VMIPS foi projetado com a propriedade de que todas as instruções de aritmética vetorial só permitem que o elemento *N* de um registrador vetorial tome parte das operações com o elemento *N* de outros registradores vetoriais. Isso simplifica bastante a construção de uma unidade vetorial altamente paralela, que pode ser estruturada como múltiplas *pistas* paralelas. Assim como em uma rodovia de trânsito, podemos aumentar a vazão de pico de uma unidade vetorial acrescentando pistas. A estrutura de uma unidade vetorial de quatro pistas aparece na Figura 4.5. Assim, mudar de uma pista para quatro pistas reduz o número de clocks de um chime de 64 para 16. Para que múltiplas pistas sejam vantajosas, as aplicações e a arquitetura devem suportar vetores longos. Caso contrário, elas serão executadas tão rapidamente que você vai ficar sem largura de banda de instrução, requerendo técnicas de ILP (Cap. 3) para fornecer instruções vetoriais suficientes.

Cada pista contém uma parte do banco de registradores vetoriais e um pipeline de execução de cada unidade funcional vetorial. Cada unidade funcional vetorial executa instruções

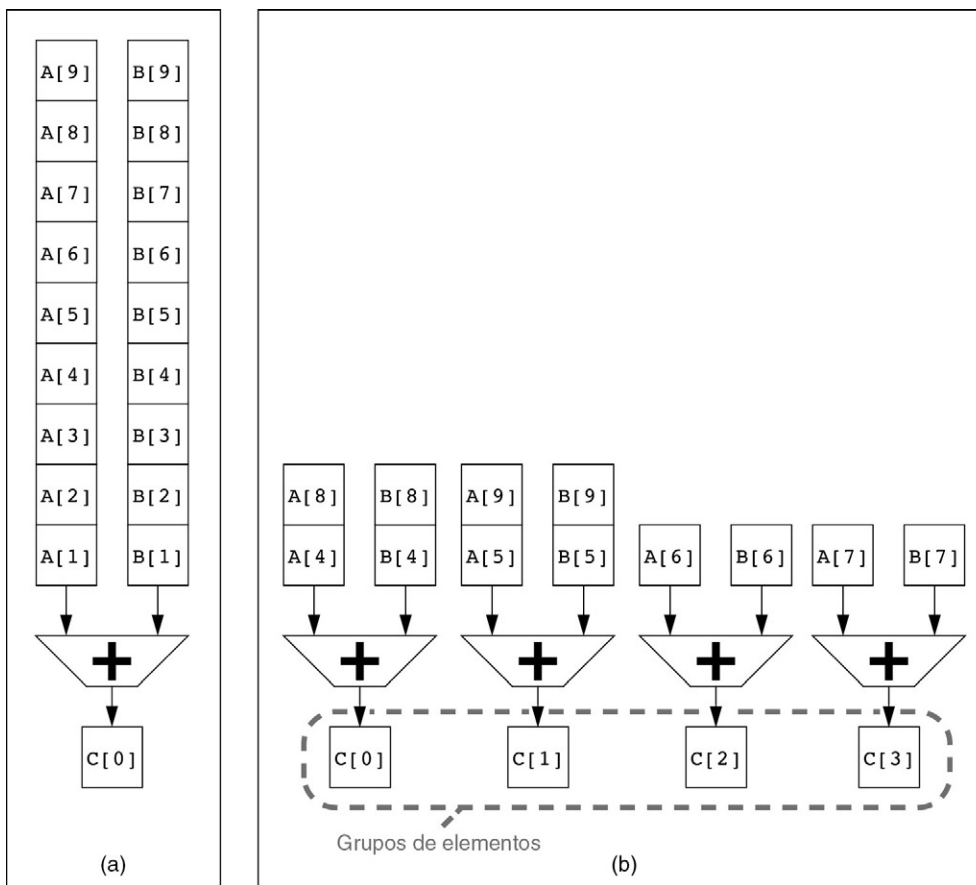



FIGURA 4.4 Uso de múltiplas unidades funcionais para melhorar o desempenho de uma única instrução de adição vetorial, C = A + B.

A máquina mostrada em (a) tem um pipeline de adição única e pode completar uma adição por ciclo. A máquina mostrada em (b) possui quatro pipelines de adição e pode completar quatro adições por ciclo. Os elementos dentro de uma única instrução de adição vetorial são intercalados pelos quatro pipelines. O conjunto de elementos que se movem pelos pipelines juntos é chamado *grupo de elementos. (Reproduzido com permissão de Asanovic, 1998.)*

vetoriais na velocidade de um grupo de elementos por ciclo usando múltiplos pipelines, um por pista. A primeira pista mantém o primeiro elemento (elemento 0) para todos os registradores vetoriais, por isso o primeiro elemento em qualquer instrução vetorial terá seus próprios operandos-fonte e destino localizados na primeira pista. Essa alocação permite que o pipeline aritmético local à pista termine a operação sem se comunicar com outras pistas. O entrelaçamento de fios entre as pistas só é necessário para o acesso à memória principal. Essa falta de comunicação entre as pistas reduz o custo de fiação e as portas do banco de registradores exigidas para a montagem de uma unidade de execução altamente paralela, e ajuda a explicar por que os supercomputadores vetoriais atuais podem completar até 64 operações por ciclo (duas unidades aritméticas e duas unidades carregamento-armazenamento por 16 pistas).

A adição de múltiplas pistas é uma técnica popular para melhorar o desempenho do vetor, pois exige pouco aumento na complexidade de controle e não exige mudanças no código de máquina existente. Isso também permite aos projetistas reduzir a área do substrato, a taxa de clock, a voltagem e a energia sem sacrificar o desempenho de pico. Se a taxa de clock de um processador vetorial for reduzida pela metade, dobrar o número de pistas vai manter o mesmo desempenho potencial.

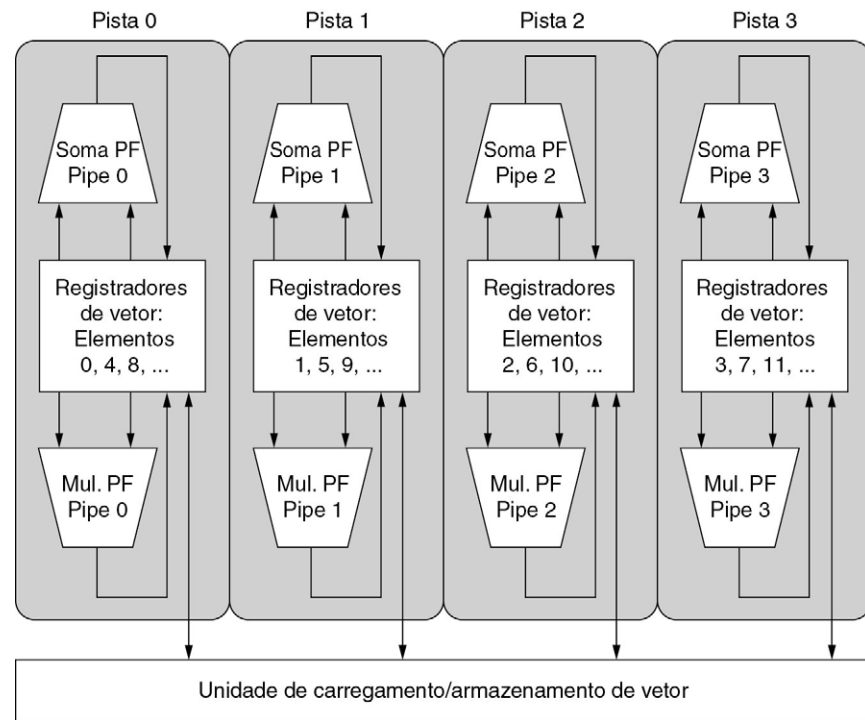



FIGURA 4.5 Estrutura de uma unidade vetorial contendo quatro pistas.
O armazenamento do registrador vetorial é dividido pelas pistas, com cada pista mantendo cada quarto elementos de cada registrador vetorial. São mostradas três unidades funcionais vetoriais, uma de adição de PF, uma de multiplicação de PF e uma unidade carregamento-armazenamento. Cada uma das unidades aritméticas vetoriais contém quatro pipelines de execução, uma por pista, que atuam em conjunto para completar uma única instrução vetorial. Observe como cada seção do banco de registradores vetorial só precisa oferecer portas suficientes para os pipelines locais à sua pista; isso reduz drasticamente o custo de fornecer múltiplas portas aos registradores vetoriais. O caminho para oferecer o operando escalar para as instruções escalares vetoriais não aparece nesta figura, mas o valor escalar precisa ser enviado a todas as pistas por broadcast.

Registradores de tamanho do vetor: tratando loops diferentes de 64

Um processador de registrador vetorial tem o tamanho vetorial natural determinado pelo número de elementos em cada registrador vetorial. Esse tamanho, que é de 64 para VMIPS, provavelmente não combina com o tamanho de vetor real em um programa. Além do mais, em um programa real, o tamanho de determinada operação vetorial normalmente é *desconhecido* durante a compilação. Na verdade, um único pedaço de código pode exigir diferentes tamanhos de vetores. Por exemplo, considere este código:

```
para (i=0; i <n; i=i+1)
     Y[i] = a * X[i] + Y[i];
```

O tamanho de todas as operações vetoriais depende de *n*, que pode nem ser conhecido antes da execução! O valor de *n* também poderia ser um parâmetro para um procedimento contendo o loop acima e, portanto, estar sujeito a mudanças durante a execução.

A solução para esses problemas é criar um *registrador de tamanho de vetor* (VLR). O VLR controla o tamanho de qualquer operação vetorial, incluindo carregamento ou armazenamento vetorial. O valor no VLR, porém, não pode ser maior que o tamanho dos registradores vetoriais. Isso resolve nosso problema desde que o tamanho real seja menor ou

igual ao *tamanho máximo de vetor* (MVL). O MVL determina o tamanho de elementos de dados em um vetor de uma arquitetura. Esse parâmetro significa que o comprimento dos registradores vetoriais pode crescer em gerações futuras de computadores sem mudar o conjunto de instruções. Como veremos na próxima seção, extensões SIMD multimídia não têm equivalente em MVL, então elas mudam o conjunto de instruções sempre que aumentam seu comprimento de vetor.

E se o valor de *n* não for conhecido durante a execução e, assim, puder ser maior que o MVL? Para enfrentar o segundo problema, em que o vetor é maior que o tamanho máximo, é usada uma técnica chamada *strip mining*. Strip mining é a geração de código de modo que cada operação vetorial seja feita para um tamanho menor ou igual ao MVL. Criamos um loop que trata de qualquer número de iterações, que seja um múltiplo do MVL, e outro loop que trate de quaisquer iterações restantes, que precisam ser menores que o MVL. Na prática, os compiladores normalmente criam um único loop strip-mined que é parametrizado para lidar com as duas partes, alterando o tamanho. Mostramos a versão strip-mined do loop DAXPY em *C*:

```
low = 0;
VL = (n % MVL); /* encontra a parte de tamanho ímpar usando módulo op % */
para (j = 0; j <= (n/MVL); j=j+1) { /*loop externo*/
      para (i = low; i < (low+VL); i=i+1) /* encontra a parte de tamanho ímpar */
            Y[i] = a * X[i] + Y[i] ; /* operação principal */
        low = low + VL; /* início do próximo vetor */
        VL = MVL; /* reinicia o tamanho com comprimento máximo do vetor */
}
```

O termo n/MVL representa o truncamento da divisão de inteiros. O efeito desse loop é bloquear o vetor em segmentos, que são então processados pelo loop interno. O tamanho do primeiro segmento é (n % MVL), e todos os segmentos subsequentes são de tamanho MVL. A Figura 4.6 mostra como dividir um vetor longo em segmentos.

O loop interno do código anterior é vetorizável com tamanho VL, que é igual a (n % MVL) ou MVL. O registrador VLR precisa ser definido duas vezes — uma em cada lugar onde a variável VL no código é atribuída.

Registradores de máscara vetorial: lidando com declarações IF em loops vetoriais

Pela lei de Amdahl, sabemos que o ganho de velocidade nos programas com níveis de vetorização baixo a moderado será muito limitado. Dois motivos pelos quais os níveis

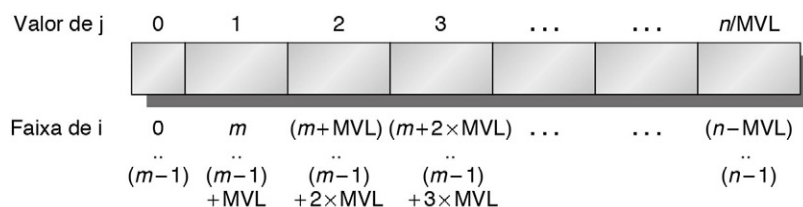



FIGURA 4.6 **Um vetor de tamanho arbitrário processado com strip mining.**

Todos os blocos, exceto o primeiro, são de tamanho MVL, utilizando a capacidade total do processador vetorial. Nesta figura, a variável *m* é usada para a expressão (n % MVL). (O operador C % é módulo.)

de vetorização mais altos não são alcançados são a presença de condicionais (instruções if) dentro dos loops e o uso de matrizes dispersas. Os programas que contêm instruções if nos loops não podem ser executados no modo vetorial usando as técnicas que discutimos até aqui, pois as instruções if introduzem dependências de controle em um loop. De modo semelhante, as matrizes dispersas não podem ser implementadas eficientemente usando qualquer uma das capacidades que vimos até agora. Discutiremos aqui as estratégias para lidar com a execução condicional, deixando a discussão das matrizes dispersas para a próxima subseção.

Considere o seguinte loop escrito em *C*:

```
para (i = 0; i < 64; i=i+1)
     if (X[i] != 0)
         X[i] = X[i] – Y[i];
```

Esse loop normalmente não pode ser vetorizado, devido à execução condicional do corpo; porém, se o loop interno pudesse ser executado para as iterações para as quais X(i) ≠ 0, então a subtração poderia ser vetorizada.

A extensão que normalmente é usada para essa capacidade é o *controle de máscara vetorial*. Os registradores de máscara fornecem, essencialmente, a execução condicional de cada operação de elemento em uma instrução vetorial. O controle de máscara vetorial usa um vetor booleano para controlar a execução de uma instrução vetorial, assim como as instruções executadas condicionalmente utilizam uma condição booleana para determinar se uma instrução é executada. Quando o *registrador de máscara vetorial* é habilitado, quaisquer instruções vetoriais executadas operam somente sobre os elementos vetoriais cujas entradas correspondentes no registrador de máscara vetorial são 1. As entradas no registrador vetorial de destino que correspondem a um 0 no registrador de máscara não são afetadas pela operação vetorial. Limpar o registrador de máscara vetorial o define com todos os bits iguais a 1, fazendo com que as instruções vetoriais subsequentes operem sobre todos os elementos do vetor. Agora o código a seguir pode ser usado para o loop anterior, supondo que os endereços de partida de *X* e *Y* estejam em Rx e Ry, respectivamente:

```
LV          V1,Rx       ;carrega vetor X em V1
LV          V2,Ry       ;carrega vetor Y
L.D         F0,#0       ;carrega zero FP em F0
SNEVS.D     V1,F0       ;define VM(i) como 1 se V1(i)!=F0
SUBVV.D     V1,V1,V2    ;subtrai sob máscara vetorial
SV          V1,Rx       ;configura a máscara de para todos iguais a 1
                        ;armazena o resultado em X
```

Os projetistas de compiladores chamam a transformação para mudar uma declaração IF em uma sequência de código de linha direta usando execução condicional de *conversão if*.

Entretanto, o uso de um registrador de máscara vetorial apresenta desvantagens. Quando examinamos instruções executadas condicionalmente, vemos que essas instruções ainda exigem tempo de execução quando a condição não é satisfeita. Apesar disso, a eliminação de um desvio e as dependências de controle associadas podem tornar uma instrução condicional mais rápida, ainda que às vezes realize trabalho inútil. Da mesma forma, as

instruções executadas com uma máscara vetorial tomam tempo de execução mesmo para os elementos onde a máscara é 0. Da mesma forma, mesmo com um número significativo de 0s na máscara, o uso do controle de máscara vetorial pode ser significativamente mais rápido do que o do modo escalar.

Como veremos na Seção 4.4, uma diferença entre os processadores vetoriais e os GPUs é o modo como eles lidam com declarações condicionais. Processadores vetoriais tornam os registradores de máscara parte do estado da arquitetura e dependem dos compiladores para manipular explicitamente os registradores de máscara. Em contraste, as GPUs obtêm o mesmo efeito usando o hardware para manipular registradores de máscara internos, que são invisíveis para o software da GPU. Nos dois casos, o hardware usa o tempo para executar um elemento vetorial quando a máscara é 0 ou 1, então a taxa GFLOPS cai quando máscaras são usadas.

Bancos de memória: fornecendo largura de banda para unidades de carregamento/armazenamento vetoriais

O comportamento da unidade vetorial de carregamento-armazenamento é significativamente mais complicado do que o das unidades funcionais aritméticas. O tempo inicial para um carregamento é o tempo para levar a primeira palavra da memória para um registrador. Se o restante do vetor puder ser fornecido sem stalls, então a taxa de iniciação do vetor será igual à taxa em que novas palavras são buscadas ou armazenadas. Diferentemente das unidades funcionais mais simples, a taxa de iniciação pode não ser necessariamente um ciclo de clock, pois os stalls do banco de memória podem reduzir o throughput efetivo.

Normalmente, as penalidades para os inícios em unidades carregamento-armazenamento são mais altas do que aquelas para as unidades funcionais aritméticas — mais de 100 ciclos de clock em alguns processadores. Para o VMIPS, consideramos um tempo de início de 12 ciclos de clock, o mesmo do Cray-1 (computadores vetoriais mais recentes usam as caches para reduzir a latência dos carregamentos e os armazenamentos vetoriais).

Para manter uma taxa de iniciação de uma palavra buscada ou armazenada por clock, o sistema de memória precisa ser capaz de produzir ou aceitar essa quantidade de dados. Isso normalmente é feito espalhando-se os acessos por vários bancos de memória independentes. Conforme veremos na próxima seção, dispor de um número significativo de bancos é útil para lidar com carregamentos ou armazenamentos vetoriais que acessam linhas ou colunas de dados.

A maioria dos processadores vetoriais utiliza bancos de memória em vez de simples intercalação por três motivos principais:

1. Muitos computadores vetoriais admitem múltiplos carregamentos ou armazenamentos por clock, e normalmente o tempo do ciclo de banco de memória é muitas vezes maior do que o tempo do ciclo de CPU. Para dar suporte a múltiplos acessos simultâneos, o sistema de memória precisa ter múltiplos bancos e ser capaz de controlar os endereços para os bancos de forma independente.
2. Conforme veremos na próxima seção, muitos processadores vetoriais admitem a capacidade de carregar ou armazenar palavras de dados que não são sequenciais. Nesses casos, é necessário haver endereçamento independente do banco em vez de intercalação.
3. Muitos computadores vetoriais admitem múltiplos processadores compartilhando o mesmo sistema de memória, por isso cada processador estará gerando seu próprio fluxo independente de endereços.

Em combinação, esses recursos levam a um grande número de bancos de memória independentes, como mostramos no exemplo a seguir.

Exemplo A maior configuração do Cray T90 (Cray T932) possui 32 processadores, cada qual capaz de gerar quatro carregamentos e dois armazenamentos por ciclo. O ciclo de clock de CPU é de 2,167 ns, enquanto o tempo de ciclo das SRAMs usadas no sistema de memória é de 15 ns. Calcule o número mínimo de bancos de memória necessários para permitir que todas as CPUs executem na largura de banda total da memória.

Resposta O número máximo de referências de memória de cada ciclo é 192 (32 CPUs vezes seis referências por CPU). Cada banco de SRAM está ocupado por 15/2,167 = 6,92 ciclos de clock, que arredondamos para sete ciclos de clock. Portanto, exigimos um mínimo de $192 \times 7 = 1.344$ bancos de memória!

O Cray T932 na realidade tem 1.024 bancos de memória, e por isso os primeiros modelos não poderiam sustentar a largura de banda total para todas as CPUs simultaneamente. Um upgrade de memória subsequente substituiu as SRAMs assíncronas de 15 ns por SRAMs síncronas em pipeline, que dividiram ao meio o tempo de ciclo de memória, fornecendo assim uma largura de banda suficiente.

Assumindo uma perspectiva de alto nível, as unidades de carregamento/armazenamento vetorial têm um papel similar ao das unidades pré-busca em processadores escalares, no sentido de que os dois tentam obter proporcional largura de banda de dados fornecendo streams de dados para os processadores.

Stride: manipulando arrays multidimensionais em arquiteuras vetoriais

A posição na memória dos elementos adjacentes em um vetor pode não ser sequencial. Considere o código *C* simples para multiplicação de matriz:

```
para (i = 0; i < 100; i=i+1)
      para (j = 0; j < 100; j=j+1) {
            A[i][j] = 0.0;
            para (k = 0; k < 100; k=k+1)
               A[i][j] = A[i][j] + B[i][k] * D[k][j];
      }
```

Poderíamos vetorizar a multiplicação de cada linha de *B* com cada coluna de *D* e realizar o strip mining do loop interno tendo *k* como variável de índice.

Para fazer isso, temos que considerar o modo como os elementos adjacentes em *B* e os elementos adjacentes em *D* são endereçados. Quando um array é alocado, ele é linearizado e precisa ser disposto em ordem de linha (em *C*) ou de coluna (como em Fortran). Essa linearização significa que os elementos na linha ou os elementos na coluna não estão adjacentes na memória. Por exemplo, o código *C* (anterior) aloca em ordem de linha, então os elementos de *D* acessados por iterações no loop interno são separados pelo tamanho da linha vezes 8 (número de bytes por entrada) para um total de 800 bytes. No Capítulo 2, vimos que o bloqueio poderia ser usado para melhorar a localidade nos sistemas baseados em cache. Para os processadores vetoriais sem caches, precisamos de outra técnica para buscar os elementos de um vetor que não estão adjacentes na memória.

Essa distância que separa os elementos que devem ser reunidos em um único registrador é chamada *passo*. Nesse exemplo, a matriz *D* tem um passo de 100 palavras duplas (800 bytes).

Quando um vetor é carregado em um registrador vetorial, atua como se tivesse elementos logicamente adjacentes. Assim, um processador vetorial pode tratar de passos maiores que 1, chamados *passos não unitários*, usando apenas operações carregamento e armazenamento vetorial com capacidade de passo. Essa capacidade de acessar locais de memória não sequenciais e remodelá-los em uma estrutura densa é uma das principais vantagens de um processador vetorial em relação a um processador baseado em cache. As caches lidam inerentemente com dados de passo unitários, de modo que, embora aumentar o tamanho de bloco possa ajudar a reduzir as taxas de perda para grandes conjuntos de dados científicos com passo unitário, aumentar o tamanho de bloco pode ter um efeito negativo para os dados que são acessados com passo não unitário. Embora as técnicas de bloqueio possam solucionar alguns desses problemas (Cap. 2), a capacidade de associar de modo eficaz dados que não são contíguos continua sendo uma vantagem para os processadores vetoriais em certos problemas, como veremos na Seção 4.7.

No VMIPS, em que a unidade endereçável é um byte, o passo para o nosso exemplo seria 800. O valor deve ser calculado dinamicamente, pois o tamanho da matriz pode não ser conhecido durante a compilação ou — assim como o tamanho do vetor — pode mudar para diferentes execuções da mesma instrução. O passo vetorial, como o endereço inicial do vetor, pode ser colocado em um registrador de uso geral. Depois, a instrução LVWS (Load Vector With Stride) do VMIPS pode ser usada para buscar o vetor em um registrador vetorial. De modo semelhante, quando um vetor de passo não unitário está sendo armazenado, SVWS (Store Vector With Stride) pode ser usada.

Complicações no sistema de memória podem ocorrer a partir do suporte a passos maiores que 1. Quando os passos não unitários são introduzidos, torna-se possível solicitar os acessos a partir do mesmo banco. Quando múltiplos acessos disputam um banco, ocorre um conflito de banco de memória e um acesso precisa ser adiado. Um conflito de banco e, portanto, um stall, ocorrerá se

$$\frac{\text{Número de bancos}}{\text{Mínimo múltiplo comum (Stride, Número de bancos)}} < \text{Tempo de ocupação do banco}$$

Exemplo Suponha que tenhamos oito bancos de memória com um tempo de ocupação do banco com seis clocks e uma latência de memória total de 12 ciclos. Quanto tempo será preciso para completar um carregamento vetorial de 64 elementos com um passo de 1? E com um passo de 32?

Resposta Como o número de bancos é maior que o tempo de ocupação do banco, para um stride de 1 o carregamento usará 12 + 64 = 76 ciclos de clock ou 1,2 clock por elemento. O pior passo possível é um valor que seja múltiplo do número de bancos de memória, como nesse caso, com um passo de 32 e oito bancos de memória. Cada acesso à memória (depois do primeiro) colidirá com o acesso anterior e terá que esperar pelo tempo de ocupado do banco de seis ciclos de clock. O tempo total será 12 + 1 + 6 * 63 = 391 ciclos de clock ou 6,1 clocks por elemento.

Gather-Scatter: lidando com matrizes dispersas em arquiteturas vetoriais

Como mencionado, matrizes dispersas são comuns e, por isso, é importante dispor de técnicas para permitir aos programas com matrizes dispersas a execução em modo vetorial. Em uma matriz dispersa, normalmente os elementos de um vetor são armazenados em alguma forma compactada e depois acessados indiretamente. Considerando uma estrutura dispersa simplificada, poderíamos ver um código semelhante a este:

```
para (i = 0; i < n; i=i+1)
     A[K[i]] = A[K[i]] + C[M[i]];
```

Esse código implementa uma soma vetorial dispersa sobre os arrays *A* e *C*, usando vetores de índice *K* e *M* para designar os elementos diferentes de zero de *A* e *C*. (*A* e *C* precisam ter o mesmo número de elementos diferentes de zero — *n* deles —, então *K* e *M* são do mesmo tamanho.)

O principal mecanismo para dar suporte a matrizes dispersas são *operações scatter-gatter* usando vetores de índice. O objetivo de tais operações é dar suporte à movimentação entre uma representação densa (ou seja, zeros não são incluídos) e a representação normal (ou seja, os zeros são incluídos) de uma matriz dispersa. Uma operação *gather* pega um *vetor de índice* e busca o vetor cujos elementos estão nos endereços dados pela adição de um endereço de base para os deslocamentos dados no vetor de índice. O resultado é um vetor não disperso em um registrador vetorial. Depois que esses elementos são operados em uma forma densa, o vetor disperso pode ser armazenado em forma expandida por um armazenamento *scatter*, usando o mesmo vetor de índice. O suporte do hardware para tais operações é chamado de *scatter-gather* e aparece em quase todos os processadores vetoriais modernos. As instruções LVI (Load Vector Indexed) e SVI (Store Vector Indexed) oferecem essas operações no VMIPS. Por exemplo, supondo que Ra, Rc, Rk e Rm contenham os endereços iniciais dos vetores na sequência anterior, o loop interno da sequência pode ser codificado com instruções vetoriais como:

```
LV          Vk, Rk            ;load K
LVI         Va, (Ra+Vk)       ;load A[K[]]
LV          Vm, Rm            ;load M
LVI         Vc, (Rc+Vm)       ;load C[M[]]
ADDVV.D     Va, Va, Vc        ;add them
SVI         (Ra+Vk), Va       ;store A[K[]]
```

Essa técnica permite que o código com matrizes dispersas seja executado no modo vetor. Um compilador de vetorização simples não poderia vetorizar automaticamente o código-fonte anterior, pois o compilador não saberia que os elementos de *K* são valores distintos e, portanto, que não existem dependências. Em vez disso, uma diretiva do programador diria ao compilador que ele poderia executar o loop no modo vetorial.

Embora carregamentos e armazenamentos indexados (gather e scatter) possam ser usados em pipeline, em geral eles rodam muito mais lentamente do que carregamentos e armazenamentos não indexados, uma vez que os bancos de memória não são conhecidos no início da instrução. Cada elemento tem um endereço individual, então eles não podem ser tratados em grupos, e pode haver conflitos em muitos locais pelos sistemas de memória. Assim, cada acesso individual incorre em latência significativa. Entretanto, como mostra a Seção 4.7, um sistema de memória pode ter melhor desempenho sendo projetado para esse caso e usando mais recursos de hardware em comparação aos casos em que os arquitetos têm uma atitude *laissez-faire* em relação a tais acessos.

Como veremos na Seção 4.4, todos os carregamentos são gathers e todos os armazenamentos são scatters nas GPUs. Para evitar a execução lenta no frequente caso dos passos unitários, cabe ao programador da GPU garantir que todos os endereços em um gather ou scatter sejam em locais adjacentes. Além disso, o hardware da GPU deve reconhecer a sequência desses endereços durante a execução para transformar os gathers e scatters no mais eficiente acesso de passo unitário à memória.

Programando arquiteturas vetoriais

Uma vantagem das arquiteturas vetoriais é que os compiladores podem dizer aos programadores, no momento da compilação, se uma seção de código será ou não vetorizada, muitas vezes dando dicas sobre o motivo de ele não ter sido vetorizado no código. Esse modelo de execução direta permite aos especialistas em outros domínios aprender como melhorar o desempenho revisando seu código ou dando dicas para o compilador quando é correto suportar a independência entre operações, como nas transferências gather-scatter de dados. É esse diálogo entre o compilador e o programador, com cada lado dando dicas para o outro sobre como melhorar o desempenho, que simplifica a programação de computadores vetoriais.

Hoje, o principal fator que afeta o sucesso com que um programa é executado em modo vetorial é a estrutura desse programa: os loops têm dependências reais de dados (Seção 4.5) ou podem ser reestruturados para não ter tantas dependências? Esse fator é influenciado pelos algoritmos escolhidos e, até certo ponto, pelo modo como eles são codificados.

Como indicação do nível de vetorização que pode ser alcançado nos programas científicos, vejamos os níveis de vetorização observados para os benchmarks Perfect Club. Esses benchmarks são aplicações científicas grandes e reais. A Figura 4.7 mostra a porcentagem das operações executadas no modo vetor para duas versões do código executando no Cray Y-MP. A primeira versão é aquela obtida apenas com otimização do compilador no código original, enquanto a segunda foi otimizada manualmente por uma equipe de programadores da Cray Research. A grande variação no nível de vetorização do compilador foi observada por vários estudos do desempenho das aplicações em processadores vetoriais.

As versões ricas em dicas mostram ganhos significativos no nível de vetorização para códigos que o compilador não poderia vetorizar bem por si só, com todos os códigos acima de 50% de vetorização. A vetorização média melhorou de aproximadamente 70% para cerca de 90%.

Nome do benchmark	Operações executadas no modo vetorial, otimizadas pelo compilador	Operações executadas no modo vetorial, com dicas de otimização	Ganho de velocidade com dicas de otimização
BDNA	96,1%	97,2%	1,52
MG3D	95,1%	94,5%	1,00
FLO52	91,5%	88,7%	N/A
ARC3D	91,1%	92,0%	1,01
SPEC77	90,3%	90,4%	1,07
MDG	87,7%	94,2%	1,49
TRFD	69,8%	73,7%	1,67
DYFESM	68,8%	65,6%	N/A
ADM	42,9%	59,6%	3,60
OCEAN	42,8%	91,2%	3,92
TRACK	14,4%	54,6%	2,52
SPICE	11,5%	79,9%	4,06
QCD	4,2%	75,1%	2,15

FIGURA 4.7 Nível de vetorização entre os benchmarks Perfect Club quando executados no Cray Y-MP (Vajapeyam, 1991).

A primeira coluna mostra o nível de vetorização obtido com o compilador, enquanto a segunda coluna mostra os resultados depois que os códigos tiverem sido otimizados à mão por uma equipe de programadores da Cray Research.

Categoria da instrução	Operandos
Soma/subtração sem sinal	Trinta e dois 8 bits, dezesseis 16 bits, oito 32 bits ou quatro 64 bits
Máximo/mínimo	Trinta e dois 8 bits, dezesseis 16 bits, oito 32 bits ou quatro 64 bits
Média	Trinta e dois 8 bits, dezesseis 16 bits, oito 32 bits ou quatro 64 bits
Deslocamento para a direita/ esquerda	Trinta e dois 8 bits, dezesseis 16 bits, oito 32 bits ou quatro 64 bits
Ponto flutuante	Dezesseis 16 bits, oito 32 bits, quatro 64 bits ou dois 128 bits

FIGURA 4.8 Resumo do suporte SIMD típico a multimídia para operações de 256 bits.
Observe que o padrão IEEE 754-2008 para ponto flutuante adicionou operações de ponto flutuante de meia precisão (16 bits) e um quarto de precisão (128 bits).

4.3 EXTENSÕES DE CONJUNTO DE INSTRUÇÕES SIMD PARA MULTIMÍDIA

Extensões SIMD multimídia começaram com a simples observação de que muitas aplicações de mídia operam com tipos de dados mais restritos do que aqueles para os quais os processadores de 32 bits foram otimizados. Muitos sistemas gráficos usavam 8 bits para representar cada uma das três cores primárias e 8 bits para transparências. Dependendo da aplicação, amostras de áudio geralmente são representadas com 8 ou 16 bits. Particionando as cadeias de carry dentro de um somador de 256 bits, um processador poderia realizar operações simultâneas em vetores curtos de 32 operandos de 8 bits, 16 operandos de 16 bits, oito operandos de 32 bits ou quatro operandos de 64 bits. O custo adicional de tais somadores particionados era baixo. A Figura 4.8 resume as instruções SIMD multimídia típicas. Como as instruções vetoriais, uma instrução SIMD especifica a mesma operação em vetores de dados, ao contrário das máquinas vetoriais com grandes arquivos de registradores, como o registrador vetorial VMIPS, que pode conter até 64 elementos de 64 bits em cada um dos registradores vetoriais. As instruções SIMD tendem a especificar menos operandos e, portanto, usam arquivos de registradores muito menores.

Em contraste com as arquiteturas vetoriais, que oferecem um conjunto elegante de instruções destinado a ser o alvo de um compilador vetorizador, as extensões SIMD têm três grandes omissões:

- As extensões multimídia SIMD fixaram o número de operandos de dados no opcode, o que levou à adição de centenas de instruções nas extensões MMX, SSE e AVX da arquitetura x86. Arquiteturas vetoriais têm um registrador de comprimento vetorial que especifica o número de operandos para a operação atual. Esses registradores vetoriais de comprimento variável acomodam facilmente programas que têm, naturalmente, vetores mais curtos do que o tamanho máximo que a arquitetura suporta. Além do mais, a arquitetura vetorial tem um comprimento máximo vetorial implícito na arquitetura, que, combinado com o registrador de comprimento vetorial, evita o uso de muitos opcodes.
- A SIMD multimídia não oferece os modos de endereçamento mais sofisticados das arquiteturas vetoriais, especificamente acessos por passo e acessos gather-scatter. Esses recursos aumentam o número de programas que um compilador vetorial pode vetorizar com sucesso (Seção 4.7).
- A SIMD multimídia geralmente não oferece os registradores de máscara para suportar a execução condicional de elementos, como nas arquiteturas vetoriais.

Essas omissões tornam mais difícil, para o compilador, gerar código SIMD e aumentam a dificuldade de programar em linguagem assembly SIMD.

Para a arquitetura x86, as instruções MMX adicionadas em 1996 modificaram o objetivo dos registradores de ponto flutuante de 64 bits, então as instruções básicas poderiam realizar oito operações de 8 bits ou quatro operações de 16 bits simultaneamente. As operações MAX e MIN se juntaram a elas, além de ampla variedade de instruções condicionais e de mascaramento, operações tipicamente encontradas em processadores de sinais digitais e instruções *ad hoc* que se acreditava serem úteis nas bibliotecas de mídia importantes. Observe que o MMX reutilizou as instruções de transferência de dados de ponto flutuante para acessar a memória.

Em 1999, as extensões SIMD para streaming (Streaming SIMD Extensions — SSE) adicionaram registradores separados que tinham 128 bits de largura, então as instruções poderiam realizar simultaneamente 16 operações de 8 bits, oito operações de 16 bits ou quatro operações de 32 bits. Ela também realizava aritmética paralela de ponto flutuante de precisão simples. Como a SSE tinha registradores separados, precisava de instruções separadas de transferência de dados. A Intel logo adicionou tipos de dados de ponto flutuante SIMD de precisão dupla através do SSE2 em 2001, SSE3 em 2004 e SSE4 em 2007. Instruções com quatro operações de ponto flutuante de precisão simples ou duas operações paralelas de precisão dupla aumentariam o pico de desempenho de ponto flutuante nos computadores ×86, contanto que os programadores colocassem os operandos lado a lado. A cada nova geração, eles também adicionaram instruções *ad hoc* cujo objetivo era acelerar funções multimídia específicas consideradas importantes.

As extensões vetoriais avançadas (Advanced Vector Extensions — AVX) adicionadas em 2010 dobraram a largura dos registradores novamente para 256 bits e, portanto, ofereceram instruções que dobraram o número de operações em todos os tipos de dados mais estreitos. A Figura 4.9 mostra instruções AVX úteis para cálculos de ponto flutuante com precisão dupla. A AVX inclui preparações para estender a largura para 512 bits e 1.024 bits nas futuras gerações da arquitetura.

Em geral, o objetivo dessas extensões tem sido acelerar bibliotecas cuidadosamente escritas em vez de o compilador gerá-las (Apêndice H), mas compiladores ×86 recentes estão tentando gerar código particularmente para aplicações intensas em termos de ponto flutuante.

Instrução AVX	Descrição
VADDPD	Soma quatro operandos de precisão dupla em pacote
VSUBPD	Subtrai quatro operandos de precisão dupla em pacote
VMULPD	Multiplica quatro operandos de precisão dupla em pacote
VDIVPD	Divide quatro operandos de precisão dupla em pacote
VFMADDPD	Multiplica e soma quatro operandos de precisão dupla em pacote
VFMSUBPD	Multiplica e subtrai quatro operandos de precisão dupla em pacote
VCMPxx	Compara quatro operandos de precisão dupla em pacote para EQ, NEQ, LT, LE, GT, GE...
VMOVAPD	Move quatro operandos alinhados de precisão dupla em pacote
VBROADCASTSD	Transmite por broadcast, um operando de precisão dupla para quatro localidades em um registrador de 256 bits

FIGURA 4.9 Instruções AVX para arquitetura ×86 úteis em programas de ponto flutuante de precisão dupla.
Duplo pacote para AVX de 256 bits significa quatro operandos de 64 bits executados em modo SIMD. Conforme a largura aumenta com o AVX, é cada vez mais importante adicionar instruções de permutação de dados que permitam combinações de operandos estreitos de diferentes partes dos registradores largos. O AVX inclui instruções que deslocam operandos de 32 bits, 64 bits ou 128 bits dentro de um registrador de 256 bits. Por exemplo, o BROADCAST replica um operando de 64 bits quatro vezes em um registrador AVX. O AVX também inclui grande variedade de instruções multiplica-soma/subtrai fundidas. Nós mostramos somente duas aqui.

Dados esses pontos fracos, por que as extensões SIMD multimídia são tão populares? 1) Elas são fáceis de adicionar à unidade aritmética padrão e de implementar; 2) elas requerem pouco estado extra em comparação com as arquiteturas vetoriais, que são sempre uma preocupação para os tempos de troca de contexto; 3) você precisa de muita largura de banda de memória para suportar uma arquitetura vetorial, o que muitos computadores não têm; 4) a SIMD não tem de lidar com problemas de memória virtual quando uma única instrução que pode gerar 64 acessos de memória pode ter uma falha de página no meio do vetor. Extensões SIMD usam transferências de dados separadas por grupo de operandos SIMD que estejam alinhados na memória, então não podem ultrapassar os limites da página. Outra vantagem dos "vetores" de tamanho fixo curtos do SIMD é que é fácil introduzir instruções que podem ajudar com novos padrões de mídia, como as que realizam permutações ou as que consomem menos ou mais operandos do que os vetores podem produzir. Por fim, havia preocupação sobre quão bem as arquiteturas vetoriais podem trabalhar com caches. Arquiteturas vetoriais mais recentes têm tratado de todos esses problemas, mas o legado de falhas passadas moldou a atitude cética dos arquitetos com relação aos vetores.

Exemplo Para dar uma ideia de como são as instruções multimídia, suponha que tenhamos adicionado instruções multimídia SIMD de 256 bits ao MIPS. Neste exemplo, nos concentramos em ponto flutuante. Nós adicionamos o sufixo "4D" às instruções que operam com quatro operandos de precisão dupla por vez. Como nas arquiteturas vetoriais, você pode pensar em um processador SIMD como tendo pistas, quatro neste caso. O MIPS SIMD vai reutilizar os registradores de ponto flutuante como operandos em instruções 4D, assim como a precisão dupla reutilizou registradores de precisão simples no MIPS original. Este exemplo mostra um código MIPS para o loop DAXPY. Suponha que os endereços de início de *X* e *Y* sejam, respectivamente, Rx e Ry. Sublinhe as mudanças no código MIPS para SIMD.

Resposta Aqui está o código MIPS:

```
      L.D      F0,a        ;Carrega o escalar a
      MOV      F1, F0      ;Copia a em F1 para SIMD MUL
      MOV      F2, F0      ;Copia a em F2 para SIMD MUL
      MOV      F3, F0      ;Copia a em F3 para SIMD MUL
      DADDIU   R4,Rx,#512  ;Último endereço a carregar
Loop: L.4D     F4,0(Rx)    ;carrega X[i], X[i+1], X[i+2], X[i+3]
      MUL.4D   F4,F4,F0    ;a×X[i],a×X[i+1], a×X [i+2], a×X [i+3]
      L.4D     F8,0(Ry)    ;carrega Y[i], Y[i+1], Y[i+2], Y[i+3]
      ADD.4D   F8,F8,F4    ; a×X [i]+Y[i], ..., a×X [i+3]+Y[i+3]
      S.4D     F8,0(Rx)    ;Armazena em Y[i], Y[i+1], Y[i+2], Y[i+3]
      DADDIU   Rx,Rx,#32   ;Incrementa o índice de X
      DADDIU   Ry,Ry,#32   ;Incrementa o índice de Y
      DSUBU    R20,R4,Rx   ;Calcula limite
      BNEZ     R20,Loop    ;Verifica se está concluído
```

As mudanças consistiram em substituir cada instrução MIPS de precisão dupla pelo seu equivalente 4D, aumentando o incremento de oito para 32, e mudar os registradores de F2 para F4 e de F4 para F8 para obter espaço suficiente no banco de registradores para quatro operandos sequenciais de precisão dupla. Para que cada pista SIMD tivesse sua própria cópia do escalar *a*, nós copiamos o valor de F0 para os registradores F1, F2 e F3 (extensões de instruções SIMD reais têm uma instrução para transmitir um valor para todos os outros registradores em um grupo). Assim, a multiplicação realiza F4*F0, F5*F1, F6*F2 e F7*F3. Embora não tão drástico quanto a redução de 100x em largura de banda de instrução dinâmica do VMIPS, o SIMD MIPS obtém uma redução de 4x: 149 *versus* 578 instruções executadas pelo MIPS.

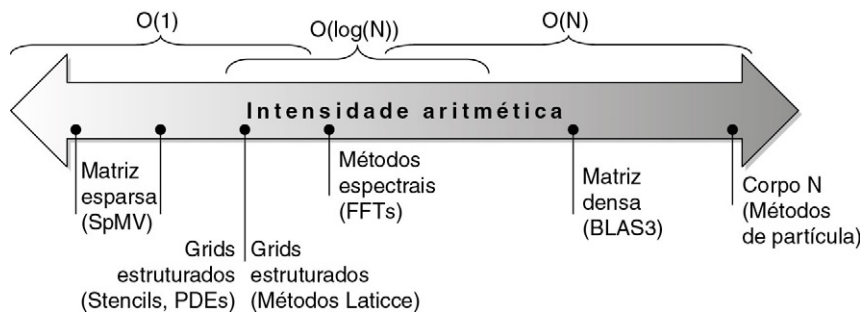



FIGURA 4.10 Intensidade aritmética especificada como o número de operações de ponto flutuante a serem executadas no programa dividido pelo número de bytes acessados na memória principal (Williams *et al.*, 2009). Alguns kernels têm uma intensidade aritmética que aumenta com o tamanho do problema, como uma matriz densa, mas há muitos kernels com intensidades aritméticas independentes do tamanho do problema.

Programando arquiteturas multimídia SIMD

Dada a natureza *ad hoc* das extensões multimídia SIMD, o modo mais fácil de usar essas instruções é por meio de bibliotecas ou escrevendo em linguagem assembly.

Extensões recentes se tornaram mais regulares, dando ao compilador um alvo mais razoável. Pegando emprestadas técnicas dos compiladores vetorizadores, os compiladores estão começando a produzir automaticamente instruções SIMD. Por exemplo, os compiladores avançados de hoje podem gerar instruções SIMD de ponto flutuante para gerar em códigos científicos com grande desempenho. Entretanto, os programadores devem ter a certeza de alinhar todos os dados na memória à largura da unidade SIMD na qual o código é executado para impedir que o compilador gere instruções escalares para código que pode ser vetorizado.

O modelo Roofline de desempenho visual

Um modo visual e intuitivo de comparar o desempenho potencial de ponto flutuante das variações da arquitetura SIMD é o modelo Roofline (Williams *et al.*, 2009). Ele liga o desempenho de ponto flutuante, o desempenho de memória e a intensidade aritmética em um gráfico bidimensional. *Intensidade aritmética* é a razão das operações de ponto flutuante por byte de memória acessado. Ela pode ser calculada tomando-se o número total de operações de ponto flutuante de um programa dividido pelo número total de bytes de dados transferidos para a memória principal durante a execução do programa. A Figura 4.10 mostra a intensidade aritmética relativa de diversos kernels de exemplo.

O pico do desempenho de ponto flutuante pode ser encontrado usando as especificações de hardware. Muitos dos kernels nesse estudo de caso são caches integradas no chip, então o pico de desempenho da memória é definido pelo sistema de memória por trás das caches. Observe que nós precisamos do pico de largura de banda de memória que está disponível para os processadores, não só nos pinos da DRAM como na Figura 4.27, na página 285. Um modo de encontrar o pico de desempenho de memória (entregue) é executar o benchmark Stream.

A Figura 4.11 mostra o modelo Roofline para o processador vetorial NEC SX-9 à esquerda e o computador multicore Intel Core i7 920 à direita. O eixo *Y* vertical é o desempenho de ponto flutuante alcançável de 2-256 GFLOP/s. O eixo *X* horizontal é a intensidade aritmética, variando de 1/8 FLOP/bytes da DRAM acessados a 16 FLOP/bytes da DRAM acessados, em ambos os gráficos. Observe que o gráfico está em escala logarítmica e que os Rooflines são feitos somente uma vez por computador.

Para um dado kernel, podemos encontrar um ponto no eixo *X*, baseado na sua intensidade aritmética. Se traçarmos uma linha vertical através desse ponto, o desempenho do kernel

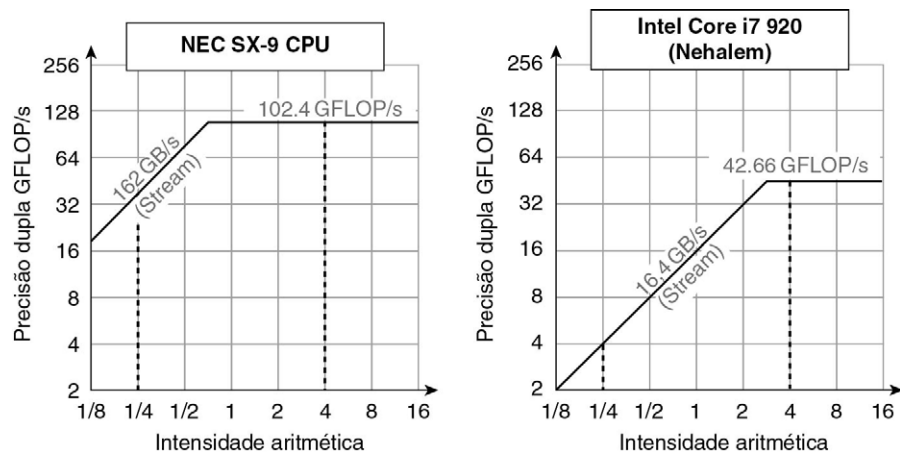



FIGURA 4.11 Modelo Roofline para um processador vetorial NEC SX-9 à esquerda e um computador multicore Intel i7 920 com extensões SIMD à direita (Williams *et al.*, 2009).

Esse Roofline é para acessos de memória de passo unitário e desempenho de ponto flutuante de precisão dupla. O NEC SX-9 é um supercomputador vetorial anunciado em 2008 que custa milhões de dólares. Ele tem um pico de desempenho PF PD de 102,4 GFLOP/s e um pico de largura de banda de memória de 162 GBytes/s do benchmark Stream. O Core i7 920 tem um pico de desempenho PF PD de 42,66 GFLOP/s e um pico de largura de banda de memória de 16,4 GBytes/s. As linhas tracejadas verticais na intensidade aritmética de 4 FLOP/byte mostram que os dois processadores operam no pico de desempenho. Nesse caso, o SX-9 a 102,4 FLOP/s é 2,4× mais rápido do que o Core i7 a 42,66 GFLOP/s. A uma intensidade aritmética de 0,25 FLOP/byte, o SX-9 é 10x mais rápido, a 40,5 FLOP/s contra 4,1 GFLOP/s para o Core i7.

nesse computador deverá estar em algum ponto ao longo dessa linha. Podemos traçar uma linha horizontal mostrando o pico de desempenho de ponto flutuante do computador. Obviamente, o real desempenho de ponto flutuante não pode ser maior do que a linha horizontal, uma vez que esse é um limite de hardware.

Como podemos traçar o pico de desempenho de memória? Uma vez que o eixo *X* é FLOP/byte e o eixo *Y* é FLOP/s, bytes/s é só uma linha diagonal com ângulo de 45 graus nessa figura. Portanto, podemos traçar uma terceira linha que nos dê o máximo desempenho de ponto flutuante que o sistema de memória desse computador pode suportar para dada intensidade aritmética. Podemos expressar os limites como uma fórmula para traçar essas linhas nos gráficos na Figura 4.11:

$$\text{GFLOPs}/s\,\text{atingível} = \text{Min}\,(\text{pico de memória BW} \times \text{intensidade aritmética}, \\ \text{pico de desempenho de ponto flutuante})$$

As linhas horizontais e verticais dão o nome a esse modelo[2] simples e indicam seu valor. O "Roofline" estabelece um limite superior sobre o desempenho de um kernel, dependendo da sua intensidade aritmética. Se pensarmos na intensidade aritmética como um mastro que atinge o telhado, ele atinge a parte plana no telhado, o que significa que o desempenho é limitado computacionalmente, ou atinge a parte inclinada do telhado, o que significa que o desempenho é limitado pela largura de banda da memória. Na Figura 4.11, a linha tracejada vertical à direita (intensidade aritmética de 4) é um exemplo do primeiro, e a linha tracejada vertical à esquerda (intensidade aritmética de 1/4) é um exemplo do segundo. Dado um modelo Roofline de um computador, você pode aplicá-lo repetidamente, uma vez que ele não varia com o kernel.

---

[2] **Nota da Tradução:** "Roofline" é uma expressão sem tradução direta em português; equivale ao "horizonte" formado pelos telhados de casas e prédios.

Observe que o "ponto limite" onde os telhados diagonal e horizontal se encontram oferece um conhecimento interessante sobre o computador. Se ele estiver muito à direita, somente kernels com intensidade aritmética muito alta poderão atingir o desempenho máximo desse computador. Se estiver muito à esquerda, praticamente qualquer kernel poderá atingir o desempenho máximo. Como veremos, esse processador vetorial tem uma largura de banda de memória muito maior e um ponto limite muito à esquerda, quando comparado com outros processadores SIMD.

A Figura 4.11 mostra que o pico de desempenho computacional do SX-9 é 2,4 vezes mais rápido do que o do Core i7, mas o desempenho de memória é 10 vezes maior. Para programas com intensidade aritmética de 0,25, o SX-9 é 10 vezes mais rápido (40,5 contra 4,1 GFLOP/s). A maior largura de banda de memória move o ponto limite de 2,6 no Core i7 para 0,6 no SX-9, o que significa que muito mais programas podem atingir o pico do desempenho computacional no processador vetorial.

4.4 UNIDADES DE PROCESSAMENTO GRÁFICO

Por poucas centenas de dólares, qualquer um pode comprar uma GPU com centenas de unidades paralelas de ponto flutuante, que tornam a computação de alto desempenho mais acessível. O interesse em computação GPU aumentou quando seu potencial foi combinado com uma linguagem de programação que tornou as GPUs mais fáceis de programar. Portanto, hoje muitos programadores de aplicações científicas e de multimídia estão ponderando se usam GPUs ou CPUs.

As GPUs e CPUs não têm um ancestral comum na genealogia das arquiteturas de computador. Não há elo perdido que explique as duas. Como descrito pela Seção 4.10, os primeiros ancestrais das GPUS são os aceleradores gráficos, já que criar gráficos bem é a razão pela qual elas existem. Embora as GPUs estejam rumando para corrente principal da computação, não podem abandonar sua responsabilidade de continuarem sendo excelentes com gráficos. Assim, seu projeto pode fazer mais sentido quando os arquitetos perguntam, dado o hardware investido para criar bons gráficos: como podemos suplementá-lo para melhorar o desempenho de uma gama maior de aplicações?

Observe que esta seção se concentra no uso das GPUs para computação. Para ver como a computação por GPU combina com o papel tradicional da aceleração de gráficos, leia "Graphics and Computing GPUs", de John Nickolls e David Kirk (Apêndice A da 4ª edição de *Computer Organization and Design*, também de nossa autoria).

Como a terminologia e alguns recursos de hardware são muito diferentes entre arquiteturas vetoriais e SIMD, acreditamos que será mais fácil se começarmos com o modelo de programação simplificada para GPUs antes de descrevermos a arquitetura.

Programando a GPU

> A CUDA é uma solução elegante para o problema de representar paralelismo em algoritmos, não em todos os algoritmos, mas o suficiente para ser importante. Ela parece ressonar, de algum modo, com o modo como pensamos e codificamos, permitindo uma expressão mais fácil e natural do paralelismo além do nível de tarefa.
>
> **Vincent Natol**
> "Kudos for CUDA", *HPC Wire* (2010)

O desafio do programador de GPU não consiste simplesmente em obter bons desempenhos da GPU, mas também em coordenar o escalonamento da computação no

processador do sistema e a GPU, assim como a transferência de dados entre a memória do sistema e a memória da GPU. Além do mais, como veremos mais adiante nesta seção, as GPUs têm quase todos os tipos de paralelismo que podem ser capturados pelo ambiente de programação: multithreading, MIMD, SIMD e mesmo em nível de instrução.

A NVIDIA decidiu desenvolver uma linguagem semelhante a *C* e um ambiente de programação que melhorariam a produtividade dos programadores de GPU, atacando os desafios da computação heterogênea e do paralelismo multifacetado. O nome do seu sistema é *CUDA*, de Arquitetura de Computação de Dispositivo Unificado (Compute Unified Device Architecture). A CUDA produz C/C++ para o processador do sistema (*host*) e um dialeto C e C + + para a GPU (*dispositivo*, daí o D em CUDA). Uma linguagem de programação similar é a OpenCL, que muitas empresas estão desenvolvendo para oferecer uma linguagem independente de fornecedor para múltiplas plataformas.

A NVIDIA decidiu que o tema unificador de todas essas formas de paralelismo é o *Thread CUDA*. Usando esse nível mais baixo de paralelismo como a primitiva de programação, o compilador e o hardware podem reunir milhares de threads CUDA para utilizar os diversos estilos de paralelismo dentro de uma GPU: multithreading, MIMD, SIMD e em nível de instrução. Assim, a NVIDIA classifica o modelo de programação CUDA como instrução única, múltiplos threads (Single Instruction, Multiple Thread — SIMT). Por motivos que veremos em breve, esses threads são reunidos em bloco e executados em grupos de 32 threads, chamados *bloco de thread.* O hardware que executa um bloco inteiro de threads é chamado *processador SIMD multithreaded.*

Precisamos somente de alguns detalhes antes de dar um exemplo de um programa CUDA:

- Para distinguir entre funções da GPU (dispositivo) e funções do processador do sistema (host), a CUDA usa `_device_` ou `_global_` para o primeiro e `_host_` para o segundo.
- As variáveis CUDA declaradas como nas funções `_device_` ou `_global_` são alocadas na memória da GPU (ver adiante), que é acessível por todos os processadores SIMD multithreaded.
- A sintaxe da função estendida de chamada para a função *name* que é executada na GPU é

```
name<<<dimGrid, dimBlock>>>(... lista de parâmetros ...)
```

onde `dimGrid` e `dimBlock` especificam as dimensões do código (em blocos) e as dimensões de um bloco (em threads).

- Além do identificador de blocos (`blockIdx`) e do identificador de threads por bloco (`threadIdx`), a CUDA fornece um indentificador para o número de threads por bloco (`blockDim`), que vem do parâmetro `dimBlock` do item anterior.

Antes de ver o código CUDA, vamos começar com um código *C* convencional para o loop DAXPY da Seção 4.2:

```
//    Invoca DAXPY
daxpy(n, 2.0, x, y);
//    DAXPY em C
void daxpy(int n, double a, double *x, double *y)
{
para (int i = 0; i < n; ++i)
       y[i] = a*x[i] + y[i];
}
```

A seguir está a versão CUDA. Lançamos *n* threads, um por elemento de vetor, com 256 threads CUDA por bloco de threads em um processador SIMD multithreaded. A função da GPU começa calculando-se o índice de elemento correspondente *i*, baseado no ID do bloco, o número de threads por bloco e o ID do thread. Enquanto esse índice estiver dentro do array (i < n), realiza a multiplicação e adição.

```
// Invoca DAXPY com 256 threads por bloco de threads
__host__
int nblocks = (n+ 255) / 256;
   daxpy<<<nblocks, 256>>>(n, 2.0, x, y);
// DAXPY em CUDA
__device__
void daxpy(int n, double a, double *x, double *y)
{
   int i = blockIdx.x*blockDim.x + threadIdx.x;
   if (i < n) y[i] = a*x[i] + y[i];
}
```

Comparando os códigos *C* e CUDA, vemos um padrão comum para paralelizar o código CUDA com dados paralelos. A versão *C* tem um loop em que cada iteração é independente das outras. Isso permite que o loop seja transformado diretamente em um código paralelo no qual cada iteração de loop se torna um thread independente. (Como mencionado antes e descrito em detalhes na Seção 4.5, compilação vetorial também depende de uma falta de dependências entre as iterações de um loop, que são chamadas *dependências carregadas em loop*.) O programador determina o paralelismo em CUDA explicitamente, especificando as dimensões do grid e o número de threads por processador SIMD. Considerando um único thread para cada elemento, não há necessidade de sincronização entre os threads quando se gravam os resultados na memória.

O hardware da GPU suporta a execução em paralelo e o gerenciamento de threads. Isso não é feito por aplicações ou pelo sistema operacional. Para simplificar o escalonamento pelo hardware, a CUDA requer que blocos de threads sejam capazes de ser executados independentemente e em qualquer ordem. Diferentes blocos de threads não podem se comunicar diretamente, embora possam se *coordenar* usando operações atômicas de memória na memória global.

Como veremos em breve, muitos conceitos de hardware de GPU não são óbvios em CUDA. Isso é uma coisa boa da perspectiva da produtividade de um programador, mas a maioria dos programadores está usando GPUs em vez de CPUs para obter desempenho. Os programadores de desempenho devem ter o hardware da GPU em mente quando escrevem em CUDA. Por motivos que explicaremos em breve, eles sabem que precisam manter juntos grupos de 32 threads no fluxo de controle para obter o melhor desempenho dos processadores SIMD multithreaded e criar muitos outros threads por processador SIMD multithreaded para ocultar a latência da DRAM. Eles também precisam manter os endereços de dados localizados em um ou alguns blocos de memória para obter o desempenho de memória esperado.

Como muitos sistemas paralelos, um compromisso entre a produtividade e o desempenho constitui a inclusão de intrínsecos na CUDA para dar aos programadores o controle explícito do hardware. Muitas vezes, a luta entre garantir a produtividade e permitir ao

programado expressar qualquer coisa que o hardware possa fazer acontece na computação paralela. Vai ser interessante ver como a linguagem evolui nessa clássica batalha entre produtividade e desempenho, além de ver se a CUDA se torna popular para outras GPUS ou mesmo outros estilos arquitetônicos.

Estruturas computacionais da GPU NVIDIA

A herança incomum mencionada ajuda a explicar por que as GPUs têm seu próprio estilo de arquitetura e sua própria terminologia, independentemente das CPUs. Um obstáculo para entender as GPUs tem sido o jargão, pois alguns termos têm até nomes enganosos. Esse obstáculo tem sido surpreendentemente difícil de superar, como podem atestar as muitas vezes que este capítulo foi reescrito. Para tentar fazer a ponte entre os objetivos gêmeos de tornar a arquitetura das GPUs compreensível *e* aprender os muitos termos de GPU com definições não tradicionais, nossa solução final foi usar a terminologia CUDA para software, mas usar inicialmente termos mais descritivos para o hardware, às vezes pegando emprestados termos usados pelo OpenCL. Depois de explicarmos a arquitetura de GPU em nossos termos, vamos relacioná-los ao jargão oficial das GPUs NVIDIA.

Da esquerda para a direita, a Figura 4.12 lista os termos mais descritivos usados nesta seção, os termos mais próximos da corrente principal da computação, os termos oficiais da GPU NVIDIA para o caso de você estar interessado e, depois, uma rápida descrição dos termos. O restante desta seção explica as características microarquiteturais das GPUs usando os termos descritivos da esquerda da figura.

Usamos sistemas NVIDIA como exemplo porque eles são representativos das arquiteturas de GPU. Seguimos especificamente a terminologia da linguagem de programação paralela CUDA acima e usamos a arquitetura de Fermi como exemplo (Seção 4.7).

Como as arquiteturas vetoriais, as GPUs funcionam bem somente com problemas paralelos em nível de dados. Os dois estilos possuem transferências de dados gather-scatter e registradores de máscara, e os processadores de GPU têm ainda mais registradores do que os processadores vetoriais. Como não possuem um processador escalar próximo, às vezes as GPUs implementam em tempo de execução, no hardware, um recurso que os computadores vetoriais implementam em tempo de compilação, em software. Ao contrário da maioria das arquiteturas vetoriais, as GPUs também dependem de multithreading dentro de um único processador SIMD multithreaded para ocultar a latência de memória (Caps. 2 e 3). Entretanto, o código eficiente tanto para arquiteturas vetoriais quanto para GPUs requer que os programadores pensem em grupos de operações SIMD.

Um *grid* é o código executado em uma GPU que consiste em um conjunto de *blocos de threads.* A Figura 4.12 traça a analogia entre um grid e um loop vetorizado e entre um bloco de threads e um corpo desse loop (depois que ele foi expandido, de modo que seja um loop computacional completo). Para dar um exemplo concreto, vamos supor que queiramos multiplicar dois vetores, cada qual com 8.192 elementos. Vamos retomar esse exemplo ao longo desta seção. A Figura 4.13 mostra o relacionamento entre ele e os dois primeiros termos de GPU. O código de GPU que trabalha sobre toda a multiplicação dos 8.192 elementos é chamado *grid* (ou loop vetorizado). Para que possa ser dividido em partes mais gerenciáveis, um grid é composto de *blocos de threads* (ou corpo de um loop vetorizado), cada qual com até 512 elementos. Observe que uma instrução SIMD executa 32 elementos por vez. Com 8.192 elementos nos vetores, esse exemplo tem 16 blocos de threads, já que 16 = 8.192/512. O grid e o bloco de threads são abstrações de programação implementadas no hardware da GPU que ajudam os programadores a organizar seu código CUDA. (O bloco de threads é análogo a um loop vetorial expandido com comprimento vetorial de 32.)

Tipo	Nome mais descritivo	Termo antigo mais próximo fora das GPUS	Termos oficiais para GPU CUDA/NVIDIA	Definição do livro
Abstrações de programa	Loop vetorizável	Loop vetorizável	Grid	Loop vetorizável, executado na GPU, composto de um ou mais blocos de threads (corpos de loop vetorizado) que podem ser executados em paralelo.
	Corpo do loop vetorizado	Corpo de um loop vetorizado (strip mined)	Bloco de threads	Loop vetorizado executado em um processador SIMD multithreaded, composto de um ou mais threads de instruções SIMD. Eles podem se comunicar através da memória local.
	Sequência de operações de pista SIMD	Uma iteração de um loop escalar	Thread CUDA	Corte vertical de um thread de instruções SIMD correspondendo a um elemento executado por uma pista SIMD. O resultado é armazenado de acordo com a máscara e o registrador de predicado.
Objeto de máquina	Thread de instruções SIMD	Thread de instruções vetoriais	Warp	Thread tradicional, mas contém somente instruções SIMD que são executadas em um processador SIMD multithreaded. Resultados armazenados dependendo de uma máscara por elemento.
	Instrução SIMD	Instrução vetorial	Instrução PTX	Uma única instrução SIMD executada através das pistas SIMD.
Hardware de processamento	Processador SIMD multithread	Processador vetorial (multithreaded)	Multiprocessador de streaming	Processador SIMD multithreaded executa threads de instruções SIMD, independentemente de outros processadores SIMD.
	Escalonador de blocos de threads	Processador escalar	Engine Giga thread	Designa múltiplos blocos de threads (corpos de loops vetorizados) a processadores SIMD multithreaded.
	Escalonador de threads SIMD	Escalonador de threads em uma CPU multithreaded	Escalonador warp	Unidade de hardware que escalona e despacha threads de instruções SIMD quando elas estão prontas para ser executadas. Inclui um scoreboard para rastrear a execução de threads SIMD.
	Pista SIMD	Pista vetorial	Processador de thread	Uma pista SIMD executa as operações em um thread de instruções SIMD em um único elemento. Os resultados são armazenados de acordo com a máscara.
Hardware de memória	Memória da GPU	Memória principal	Memória global	Memória DRAM acessível por todos os processadores SIMD multithreaded em uma GPU.
	Memória privativa	Pilha ou armazenamento local de thread (OS)	Memória local	Parte da memória DRAM privativa a cada pista SIMD.
	Memória local	Memória local	Memória compartilhada	SRAM local rápida para um processador SIMD multithreaded, indisponível para outros processadores SIMD.
	Registradores de pista SIMD	Registradores vetoriais de pista	Registradores de processador de thread	Registradores em uma única pista SIMD alocados através de todo um bloco de threads (corpo de loop vetorizado).

FIGURA 4.12 **Guia rápido para os termos de GPU usados neste capítulo.**

Usamos a primeira coluna para os termos de hardware. Quatro grupos reúnem esses 11 termos. De cima para baixo: abstrações de programa, objetos de máquina, hardware de processamento e hardware de memória. A Figura 4.21, na página 270, associa os termos vetoriais com os termos mais próximos aqui, e a Figura 4.24, na página 275, e a Figura 4.25, na página 276, revelam os termos e definições oficiais CUDA/NVIDIA e AMD, juntamente com os termos usados pelo OpenCL.

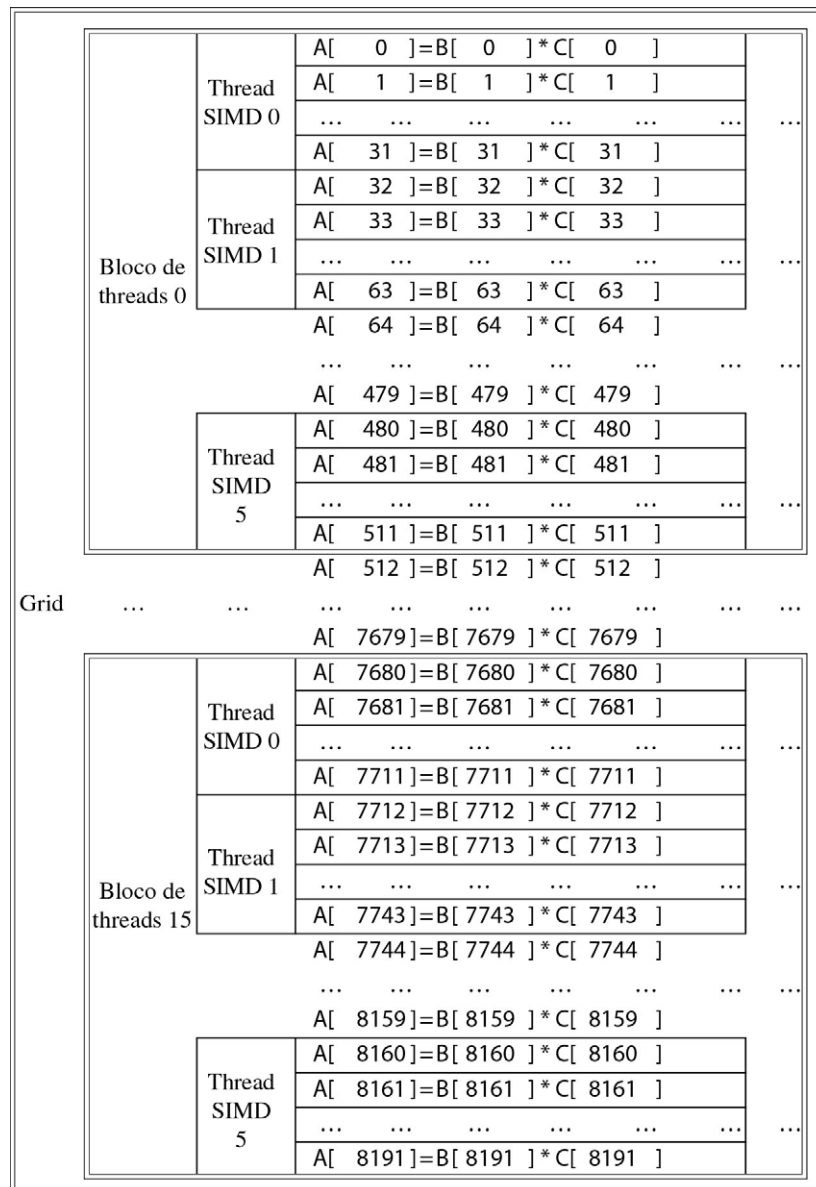



FIGURA 4.13 Mapeamento de um grid (loop vetorizável), blocos de thread (blocos SIMD básicos) e threads de instruções SIMD para uma multiplicação vetor-vetor, com cada vetor tendo 8.192 elementos.
Cada thread de instruções SIMD calcula 32 elementos por instrução, e neste exemplo cada bloco de threads contém 16 threads de instruções SIMD e o grid contém 16 blocos de threads. O hardware escalonador de bloco de threads atribui blocos de threads a processadores SIMD multithreaded e o hardware escalonador de thread seleciona qual thread de instruções SIMD executar a cada ciclo de clock em um processador SIMD. Somente threads SIMD no mesmo bloco de threads podem se comunicar através da memória local. (O número máximo de threads SIMD que podem ser executados simultaneamente por bloco de thread é 16 para GPUS da geração Tesla e 32 para as GPUs geração Fermi mais recentes.)

Um bloco de threads é designado para um processador que executa esse código, que chamamos *processador SIMD multithreaded*, pelo *escalonador de bloco de threads*. Esse escalonador tem algumas similaridades com um processador de controle em uma arquitetura vetorial. Ele determina o número de blocos de threads necessários para o loop e continua a alocá-los para diferentes processadores SIMD multithreaded até que o loop seja completado. Neste

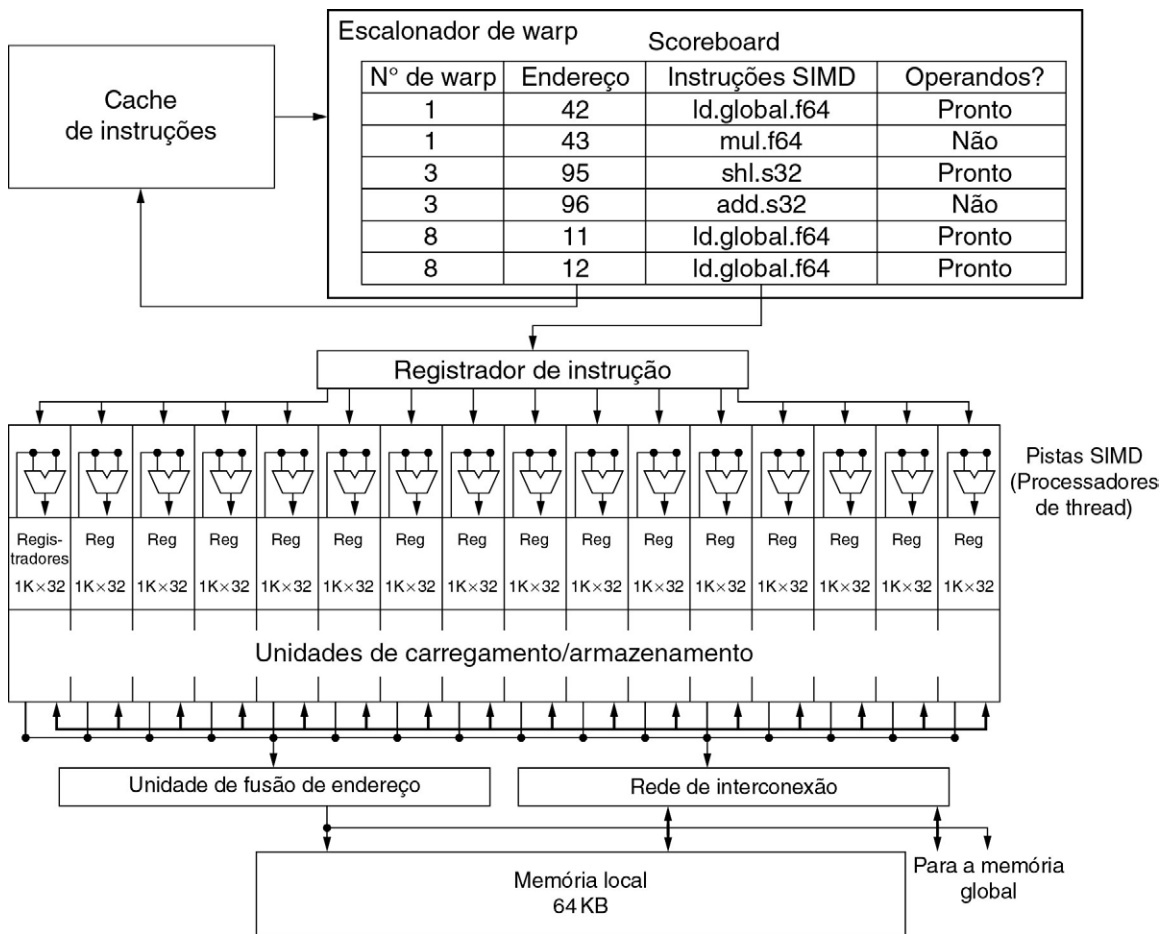



FIGURA 4.14 **Diagrama de blocos simplificados de um processador SIMD multithreaded.**
Ele tem 16 pistas SIMD. O escalonador de threads SIMD tem cerca de 48 threads independentes de instruções SIMD, que ele escalona com uma tabela de 48 PCs.

exemplo, ele enviaria 16 blocos de threads para processadores SIMD multithreaded a fim de calcular os 8.192 elementos desse loop.

A Figura 4.14 mostra um diagrama de blocos simplificado de um processador SIMD multithreaded. Ele é similar a um processador vetorial, mas tem muitas unidades funcionais paralelas, em vez de poucas que são fortemente pipelined, como em um processador vetorial. No exemplo de programação dado na Figura 4.13, cada processador SIMD multithreaded recebe 512 elementos dos vetores para trabalhar. Os processadores SIMD são processadores completos com PCs separados e programados usando threads (Cap. 3).

O hardware de GPU contém, portanto, uma coleção de processadores SIMD multithreaded que executam um grid de blocos de threads (corpos de loop vetorizado). Ou seja, uma GPU é um multiprocessador composto de processadores SIMD multithreaded.

As quatro primeiras implementações da arquitetura Fermi tinham sete, 11, 14 e 15 processadores SIMD multithreaded. Versões futuras poderão ter somente duas ou quatro. Para fornecer escalabilidade transparente entre modelos de GPUs com números diferentes de processadores SIMD multithreaded, o escalonador de blocos de threads designa blocos de thread (corpos de um loop vetorizado) a processadores SIMD multithreaded. A Figura 4.15 mostra a planta da implementação GTX 480 da arquitetura Fermi.

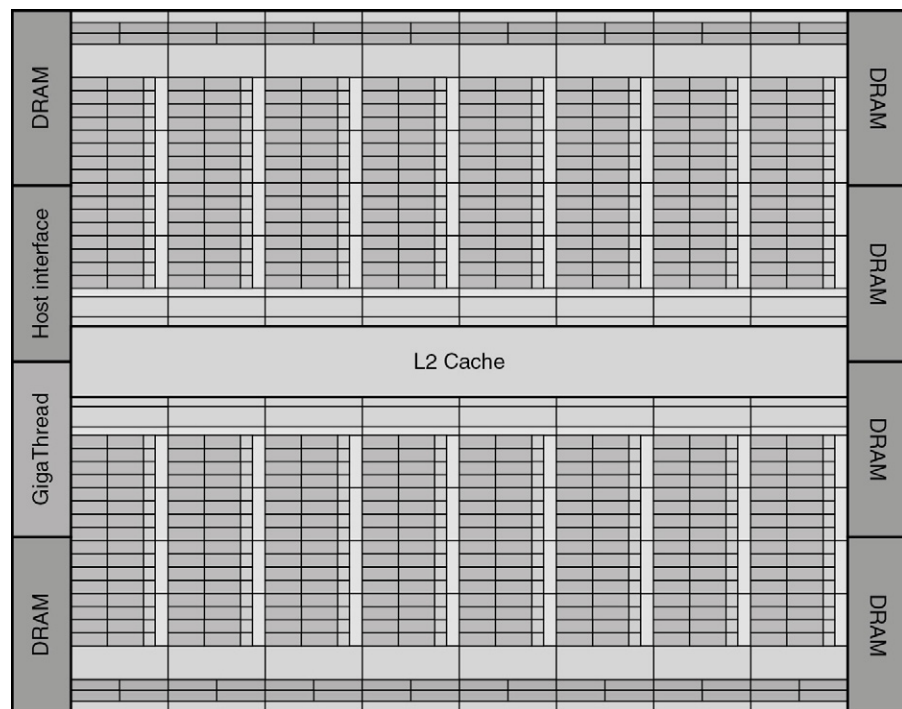



FIGURA 4.15 Planta da GPU Fermi GTX 480.

Este diagrama mostra 16 processadores SIMD multithreaded. O escalonador de bloco de threads é destacado à esquerda. O GTX 480 tem seis portas GDDR5, cada uma com 64 bits de largura, suportando até 6 GB de capacidade. A interface de host é PCI Express 2.0 × 16. Giga Thread é o nome do escalonador que distribui blocos de threads para múltiplos processadores, cada qual com seu próprio escalonador de threads SIMD.

Descendo mais um nível de detalhes, o objeto de máquina que o hardware cria, gerencia, escalona e executa é um *thread de instruções SIMD*. Ele é um thread tradicional que contém exclusivamente instruções SIMD. Esses threads de instruções SIMD têm seus próprios PCs e são executados em um processador SIMD multithreaded. O *escalonador de thread SIMD* inclui um scoreboard que lhe permite saber quais threads de instruções SIMD estão prontas para serem executadas; então ele as envia para uma unidade de despacho para serem executadas no processador SIMD multithreaded. Ele é idêntico a um escalonador de thread de hardware em um processador multithreaded tradicional (Cap. 3), já que está escalonando threads de instruções SIMD. Assim, o hardware de GPU tem dois níveis de escalonadores de hardware: 1) o *escalonador de bloco de threads* que designa blocos de threads (corpos de loops vetorizados) a processadores SIMD multithreaded, o que garante que os blocos de threads sejam designados para os processadores cujas memórias locais tenham os dados correspondentes; 2) o escalonador de threads SIMD *dentro* de um processador SIMD, que escalona quando os threads de instruções SIMD devem ser executados.

As instruções SIMD desses threads têm 32 de largura, então cada thread de instruções SIMD neste exemplo calcularia 32 dos elementos do cálculo. Neste exemplo, os blocos de threads conteriam 512/32 = 16 threads SIMD (Fig. 4.13).

Uma vez que o thread consiste em instruções SIMD, o processador SIMD deve ter unidades funcionais paralelas para realizar a operação. Nós as chamamos *pistas SIMD*, e elas são bastante similares às pistas vetoriais da Seção 4.2.

O número de pistas por processador SIMD varia de acordo com as gerações de GPU. Com a Fermi, cada thread de instruções SIMD com 32 de largura é mapeado em 16 pistas SIMD físicas, então cada instrução SIMD em um thread de instruções SIMD leva dois ciclos de clock para ser completada. Cada thread de instruções SIMD é executado em *lock step* e escalonado somente no início. Continuando com a analogia de um processador SIMD como processador vetorial, você poderia dizer que ele tem 16 pistas, o comprimento do vetor seria de 32 e o chime de dois ciclos de clock. (Essa natureza ampla mas rasa é o porquê de usarmos o termo processador SIMD em vez de processador vetorial, já que ele é mais descritivo.)

Já que, por definição, os threads de instruções SIMD são independentes, o escalonador de threads SIMD pode escolher qualquer thread de instruções SIMD que esteja pronto, e não precisa se prender à próxima instrução SIMD na sequência dentro de um thread. O escalonador de threads SIMD inclui um scoreboard (Cap. 3) para rastrear até 48 threads de instruções SIMD a fim de verificar qual instrução SIMD está pronta. Esse scoreboard é necessário porque as instruções de acesso à memória podem ocupar um número imprevisível de ciclos de clock, devido a conflitos de banco de memória, por exemplo. A Figura 4.16 mostra o escalonador de threads SIMD selecionando threads de instruções SIMD em ordem diferente ao longo do tempo. A suposição dos arquitetos de GPU é de que as aplicações de GPU têm tantos threads de instruções SIMD que o multithreading pode tanto ocultar a latência da DRAM quanto aumentar a utilização de processadores SIMD multithreaded. Entretanto, para cobrir as apostas deles, a recente GPU NVIDIA Fermi inclui um cache L2 (Seção 4.7).

Continuando com nosso exemplo de multiplicação de vetores, cada processador SIMD multithreaded deve carregar 32 elementos de dois vetores da memória em registradores, realizando a multiplicação lendo e gravando registradores, e armazenando o produto dos registradores de volta para a memória. Para conter esses elementos de memória, um processador SIMD tem impressionantes 32.768 registradores de 32 bits. Assim como um processador vetorial, esses registradores são divididos logicamente ao longo das pistas vetoriais ou, nesse caso, pistas SIMD. Cada thread SIMD é limitado a não mais de 64 registradores, então você pode pensar em um thread SIMD como tendo até 64 registradores vetoriais, com cada registrador tendo 32 elementos e cada elemento tendo 32 bits de largura. (Como os operandos de ponto flutuante com precisão dupla usam dois registradores adjacentes de 32 bits, uma visão alternativa é que cada thread SIMD tem 32 registradores vetoriais de 32 elementos, cada qual com 64 bits de largura.)

FIGURA 4.16
Escalonamento de threads de instruções SIMD.
O escalonador seleciona um thread de instruções SIMD pronto e despacha uma instrução sincronizada para todas as pistas SIMD, executando o thread SIMD. Já que os threads de instruções SIMD são independentes, o escalonador pode selecionar um thread SIMD diferente a cada vez.

Como o Fermi tem 16 pistas SIMD físicas, cada uma contém 2.048 registradores. (Em vez de tentar projetar registradores de hardware com muitas portas de leitura e portas de gravação por bit, as GPUs vão usar estruturas de memória mais simples, porém dividindo-as em bancos para obter largura de banda suficiente, assim como fazem os processadores vetoriais.) Cada thread CUDA obtém um elemento de cada um dos registradores vetoriais. Para lidar com os 32 elementos de cada thread de instruções SIMD com 16 pistas SIMD, os threads CUDA de um bloco de threads podem usar coletivamente até metade dos 2.048 registradores.

Para ser capaz de executar vários threads de instruções SIMD, cada conjunto de registradores é alocado dinamicamente em um processador SIMD; threads de instruções SIMD são criados e liberados quando existe o thread SIMD.

Observe que um thread CUDA é somente um corte vertical de um thread de instruções SIMD, correspondendo a um elemento executado por uma pista SIMD. Saiba que threads

CUDA são muito diferentes de threads POSIX. Você não pode fazer chamadas arbitrárias do sistema a partir de um thread CUDA.

Agora estamos prontos para ver como são as instruções de GPU.

Arquitetura de conjunto de instruções da GPU NVIDIA

A contrário da maioria dos processadores de sistema, o alvo do conjunto de instruções dos compiladores NVIDIA é uma abstração do conjunto de instruções do hardware. A *execução de threads em paralelo* (Parallel Thread Exectution — PTX) fornece um conjunto estável de instruções, além de compatibilidade ao longo das gerações de GPUs. O conjunto de instruções de hardware é ocultado do programador. As instruções PTX descrevem as operações em um único thread CUDA e, geralmente, têm correspondência um a um com as instruções de hardware, mas um PTX pode ser expandido para muitas instruções de máquina e vice-versa. A PTX usa registradores virtuais, o compilador descobre de quantos registradores físicos vetoriais um thread SIMD precisa e, então, um otimizador divide o armazenamento em um registrador disponível entre os threads SIMD. Esse otimizador também elimina o código morto, reúne instruções e calcula locais onde os desvios podem divergir e locais onde caminhos divergentes podem convergir.

Embora haja alguma similaridade entre as microarquiteturas ×86 e o PTX, no sentido de que as duas se traduzem em uma forma interna (microinstruções para o ×86), a diferença é que essa tradução acontece no hardware em runtime durante a execução no ×86 e no software e no tempo de carregamento em uma GPU.

O formato de uma instrução PTX é

```
opcode.type d, a, b, c;
```

onde *d* é o operando de destino, *a*, *b* e *c* são os operandos de origem, e o tipo (type) é um dos seguintes:

Tipo	Especificador do tipo
Bits sem tipo de 8, 16, 32 e 64 bits	.b8, .b16, .b32, .b64
Inteiro sem sinal de 8, 16, 32 e 64 bits	.u8, .u16, .u32, .u64
Inteiro com sinal de 8, 16, 32 e 64 bits	.s8, .s16, .s32, .s64
Ponto flutuante de 16, 32 e 64 bits	.f16, .f32, .f64

Os operandos de origem são registradores de 32 ou 64 bits ou um valor constante. Destinos são registradores, exceto para instruções de armazenamento.

A Figura 4.17 mostra o conjunto básico de instruções PTX. Todas as instruções podem ser previstas por 1 bit dos registradores de predicado, que pode ser configurado por uma instrução de predicado de conjunto (setp). As instruções de fluxo de controle são funções de `chamada` e `retorno`, `saída` de thread, `desvio` e sincronização de barreira para threads dentro de um bloco de threads (`bar.sync`). Colocar um predicado em frente a uma instrução de desvio nos dá desvios condicionais. O compilador ou programador PTX declara registradores virtuais como valores de 32 bits ou 64 bits com tipo ou sem tipo. Por exemplo, R0, R1, ... são para valores de 32 bits, e `RD0, RD1`,... são para registradores de 64 bits. Lembre-se de que a designação de registradores virtuais para registradores físicos ocorre no momento do carregamento no PTX.

A sequência de instruções PTX a seguir é para uma iteração do nosso loop DAXPY, da página 252:

Grupo	Instrução	Exemplo	Significado	Comentários
Aritmético	arithmetic .type = .s32, .u32, .f32, .s64, .u64, .f64			
	add.type	add.f32 d, a, b	d = a + b;	
	sub.type	sub.f32 d, a, b	d = a - b;	
	mul.type	mul.f32 d, a, b	d = a * b;	
	mad.type	mad.f32 d, a, b, c	d = a * b + c;	Multiplicação-adição
	div.type	div.f32 d, a, b	d = a / b;	Múltiplas microinstruções
	rem.type	rem.u32 d, a, b	d = a % b;	Resto inteiro
	abs.type	abs.f32 d, a	d = |a|;	
	neg.type	neg.f32 d, a	d = 0 - a;	
	min.type	min.f32 d, a, b	d = (a < b)? a:b;	Flutuante seleciona não NaN
	max.type	max.f32 d, a, b	d = (a > b)? a:b;	Flutuante seleciona não NaN
	setp.cmp.type	setp.lt.f32 p, a, b	p = (a < b);	Compara e configura predicado
	numeric .cmp = eq, ne, lt, le, gt, ge; unordered cmp = equ, neu, ltu, leu, gtu, geu, num, nan			
	mov.type	mov.b32 d, a	d = a;	Move
	selp.type	selp.f32 d, a, b, p	d = p? a: b;	Seleciona com predicado
	cvt.dtype.atype	cvt.f32.s32 d, a	d = convert(a);	Converte atype para dtype
Função Especial	special .type = .f32 (some .f64)			
	rcp.type	rcp.f32 d, a	d = 1/a;	Recíproco
	sqrt.type	sqrt.f32 d, a	d = sqrt(a);	Raiz quadrada
	rsqrt.type	rsqrt.f32 d, a	d = 1/sqrt(a);	Raiz quadrada recíproca
	sin.type	sin.f32 d, a	d = sin(a);	Seno
	cos.type	cos.f32 d, a	d = cos(a);	Cosseno
	lg2.type	lg2.f32 d, a	d = log(a)/log(2)	Logaritmo binário
	ex2.type	ex2.f32 d, a	d = 2 ** a;	Exponencial binários
Lógico	logic.type = .pred,.b32, .b64			
	and.type	and.b32 d, a, b	d = a & b;	
	or.type	or.b32 d, a, b	d = a | b;	
	xor.type	xor.b32 d, a, b	d = a ^ b;	
	not.type	not.b32 d, a, b	d = ~a;	Complemento de um
	cnot.type	cnot.b32 d, a, b	d = (a==0)? 1:0;	Não lógico C
	shl.type	shl.b32 d, a, b	d = a << b;	Desloca para a esquerda
	shr.type	shr.s32 d, a, b	d = a >> b;	Desloca para a direita
Acesso à memória	memory.space = .global, .shared, .local, .const; .type = .b8, .u8, .s8, .b16, .b32, .b64			
	ld.space.type	ld.global.b32 d, [a+off]	d = *(a+off);	Carrega do espaço de memória
	st.space.type	st.shared.b32 [d+off], a	*(d+off) = a;	Armazena no espaço de memória
	tex.nd.dtyp.btype	tex.2d.v4.f32.f32 d, a, b	d = tex2d(a, b);	Busca de textura
	atom.spc.op.type	atom.global.add.u32 d,[a], b atom.global.cas.b32 d,[a], b, c	atomic { d = *a; *a = op(*a, b); }	Operaçãção de leitura-modificação-escrita atômica
	atom.op = and, or, xor, add, min, max, exch, cas; .spc = .global; .type = .b32			
Fluxo de controle	branch	@p bra target	if (p) goto target;	Desvio condicional
	call	call (ret), func, (params)	ret = func(params);	Chamada de função
	ret	ret	return;	Retorno da chamada de função
	bar.sync	bar.sync d	wait for threads	Sincronização de barreira
	exit	exit	exit;	Encerra execução do thread

FIGURA 4.17 Instruções de thread básicas da GPI PTX.

```
shl.u32 R8, blockIdx, 9        ;ID do Bloco de Threads*Tamanho do Bloco (512 ou 2^9)
add.u32 R8, R8, threadIdx      ;R8 = i = ID do meu thread CUDA
shl.u32 R8, R8, 3              ;offset de byte
ld.global.f64 RD0, [X+R8]      ;RD0 = X[i]
ld.global.f64 RD2, [Y+R8]      ;RD2 = Y[i]
mul.f64 RD0, RD0, RD4          ;Produto em RD0 = RD0 * RD4 (escalar a)
dd.f64 RD0, RD0, RD2           ;Soma em RD0 = RD0 + RD2 (Y[i])
st.global.f64 [Y+R8], RD0      ;Y[i] = soma (X[i]*a + Y[i])
```

Como demonstrado, o modelo de programação CUDA designa um thread CUDA para cada iteração de loop e oferece um número identificador único para cada bloco de threads (blockIdx) e um para cada thread CUDA dentro de um bloco (threadIdx). Assim, ele cria 8.192 threads CUDA e usa o número único para endereçar cada elemento no array, de modo que não há código de incremento ou desvio. As três primeiras instruções PTX calculam o offset de byte desse elemento único em R8, que é somado à base dos arrays. As instruções PTX a seguir carregam dois operandos de ponto flutuante de precisão dupla, os multiplicam e somam, e armazenam a soma. (Vamos descrever o código PTX correspondente ao código CUDA "`if (i < n)`" a seguir.)

Observe que, ao contrário das arquiteturas vetoriais, as GPUs não têm instruções separadas para transferências sequenciais de dados, transferências de dados com passo e transferências de dados gather-scatter. Todas as transferências de dados são gather-scatter! Para recuperar a eficiência das transferências de dados sequenciais (passo unitário), as GPUs incluem hardware de aglutinação de endereço para reconhecer quando as pistas SIMD dentro de um thread de instruções SIMD estão enviando coletivamente endereços sequenciais. Então, esse hardware de runtime notifica a unidade de interface de memória para requerer uma transferência de bloco de 32 palavras sequenciais. Para obter essa importante melhoria no desempenho, o programador da GPU deve garantir que threads CUDA adjacentes acessem endereços próximos ao mesmo tempo que podem ser aglutinados em um ou poucos blocos da memória ou da cache, o que nosso exemplo faz.

Desvios condicionais em GPUs

Assim como no caso das transferências de dados de passo unitário, existem fortes similaridades entre o modo como as arquiteturas vetoriais e as GPUs lidam com declarações IF: as primeiras implementam o mecanismo principalmente em software com suporte limitado de hardware e as segundas com o uso de mais hardware. Como veremos, além de usar registradores explícitos de predicados, o hardware de desvio de GPU usa máscaras internas, uma pilha de sincronização de desvio e marcadores de instrução para gerenciar quando um desvio diverge em múltiplos caminhos de execução e quando os caminhos convergem.

No nível de assembler do PTX, o fluxo de controle de um thread CUDA é descrito pelas instruções PTX branch, call, return e exit, e pelo uso de predicados individuais por pista de thread em cada instrução, especificados pelo programador com registradores de predicado com 1 bit por pista de thread. O assembler PTX analisa o gráfico de desvios PTX e o otimiza para a sequência de instruções de hardware GPU mais rápida.

No nível de instrução de hardware de GPU, o fluxo de controle inclui desvios, saltos, saltos indexados, chamadas, chamadas indexadas, retornos, saídas e instruções especiais que gerenciam a pilha de sincronização de desvio. O hardware da GPU fornece a cada thread SIMD sua própria pilha. Uma entrada de pilha contém um token identificador,

um endereço de instrução-alvo e uma máscara-alvo de thread ativo. Existem instruções especiais de GPU que empilham entradas de pilha para um thread SIMD e instruções especiais e marcadores de instrução que desempilham uma entrada de pilha ou retornam a pilha para uma entrada específica e a desviam para o endereço da instrução-alvo com a máscara-alvo de thread ativo. As instruções de hardware de GPU também possuem predicados individuais por pista (habilita/desabilita) especificados com um registrador de predicado com 1 bit para cada pista.

O assembler PTX geralmente otimiza uma simples declaração IF/THEN/ELSE de nível externo, codificada com instruções de desvio PTX em instruções de GPU com predicados, sem quaisquer instruções de desvio de GPU. Um fluxo de controle mais complexo geralmente resulta em uma mistura de predicados e instruções de desvio de GPU com instruções especiais e marcadores que usam a pilha de sincronização de desvio para empilhar uma entrada de pilha, quando algumas pistas desviam para o endereço-alvo enquanto outras caem. A NVIDIA diz que um desvio *diverge* quando isso acontece. Essa mistura também é usada quando uma pista SIMD executa um marcador de sincronização ou *converge*, que desempilha uma entrada de pilha e a desvia para o endereço de entrada de pilha com a máscara de thread ativo em nível de pilha.

O assembler PTX identifica desvios de loop e gera instruções de desvio de GPU que desviam para o topo do loop, além de instruções especiais de pilha para lidar com pistas individuais saindo do loop e convergindo as pistas SIMD quando todas tiverem completado o loop. Instruções de salto indexado e chamada indexada de GPU empilham entradas na pilha para que, quando todas as pistas completarem a declaração de troca ou chamada de função, o thread SIMD convirja.

Uma instrução de GPU configura predicado (setp na figura anterior), avaliando a parte não condicional da declaração IF. Por isso, a instrução de desvio PTX depende desse predicado. Se o assembler PTX gerar instruções com predicado sem instruções de desvio de GPU, ele usará um registrador de predicado por pista para habilitar ou desabilitar uma pista SIMD para cada instrução. As instruções SIMD nos threads dentro da parte THEN da declaração IF transmitem as operações para todas as pistas SIMD. Essas pistas com o conjunto de predicados configurado em 1 realizam a operação e armazenam o resultado, e as outras pistas SIMD não realizam uma operação ou armazenam um resultado. Para a declaração ELSE, as instruções usam o complemento do predicado (relativo à declaração THEN), então as pistas SIMD que estavam ociosas realizam a operação e armazenam o resultado, enquanto suas irmãs, que antes estavam ativas, não o fazem. No final da declaração ELSE, as instruções não possuem predicados, portanto o cálculo original pode prosseguir. Assim, para caminhos de comprimento igual, um IF-THEN-ELSE opera com 50% de eficiência.

Declarações IF podem ser aninhadas, daí o uso de uma pilha, e o assembler PTX geralmente gera uma mistura de instruções com predicado e desvios de GPU e instruções especiais de sincronização para controle complexo de fluxo. Observe que o aninhamento profundo pode significar que a maioria das pistas SIMD está ociosa durante a execução de declarações condicionais aninhadas. Assim, declarações IF duplamente aninhadas com caminhos de mesmo tamanho rodam com 25% de eficiência, triplamente aninhadas a 12,5% de eficiência, e assim por diante. O caso análogo seria o de um processador vetorial operando onde somente alguns dos bits de máscara são 1.

Descendo um nível de detalhe, o assembler PTX configura um marcador de "sincronização de desvio" em instruções condicionais apropriadas, que empilham a máscara ativa atual em uma pilha dentro de cada thread SIMD. Se o desvio condicional diverge (algumas pistas

tomam o desvio, outras não), empilha uma entrada de pilha e configura a máscara interna ativa atual com base na condição. Um marcador de sincronização de desvio desempilha a entrada de desvio que divergiu e alterna os bits de máscara antes da porção ELSE. No final da declaração IF, o assembler PTX adiciona outro marcador de sincronizador de desvio, que transfere a máscara ativa anterior da pilha para a máscara ativa atual.

Se todos os bits de máscara forem configurados para 1, então a instrução de desvio no final do THEN pula as instruções na parte ELSE. Existe uma otimização similar para a parte THEN no caso de todos os bits de máscara serem zero, já que o desvio condicional salta sobre as instruções THEN. Muitas vezes, declarações IF paralelas e desvios PTX usam condições de desvio unânimes (todas as pistas concordam em seguir o mesmo caminho), de modo que o thread SIMD não diverge em fluxos diferentes de controle de pista individual. O assembler PTX otimiza tais desvios para pular blocos de instruções que não são executados por qualquer pista de um thread SIMD. Essa otimização é útil na verificação de condição de erro, por exemplo, em que o teste deve ser feito mas raramente o desvio é tomado.

O código para uma declaração condicional similar à da Seção 4.2 é

```
if (X[i] != 0)
  X[i] = X[i] - Y[i];
else X[i] = Z[i];
```

Essa declaração IF poderia ser compilada com as seguintes instruções PTX (supondo que R8 já tenha o ID escalado do thread), com **Push*, **Comp*, **Pop* indicando os marcadores de sincronização de desvio inseridos pelo assembler PTX que empilham a máscara velha, complementam a máscara atual e desempilham para restaurar a máscara velha:

```
ld.global.f64 RD0, [X+R8]        ; RD0 = X[i]
setp.neq.s32 P1, RD0, #0         ; P1 é o registrador de predicado 1
@!P1, bra ELSE1, *Push           ; Empilha máscara velha, configura novos bits de máscara
                                 ; Se P1 for falso, vais para ELSE1
ld.global.f64 RD2, [Y+R8]        ; RD2 = Y[i]
sub.f64 RD0, RD0, RD2            ; Diferença em RD0
st.global.f64 [X+R8], RD0        ; X[i] = RD0
@P1, bra ENDIF1, *Comp           ; Bits de máscara complementar
                                 ; Se P1 for verdadeiro, vá para ENDIF1
ELSE1: ld.global.f64 RD0, [Z+R8] ; RD0 = Z[i]
       st.global.f64 [X+R8], RD0 ; X[i] = RD0
ENDIF1: <próxima instrução>, *Pop ;Desempilha para restaurar máscara antiga
```

Normalmente, todas as instruções na declaração IF-THEN-ELSE são executadas por um processador SIMD. Apenas algumas das pistas SIMD são habilitadas para as instruções THEN e outras para as instruções ELSE. Como mencionado, no caso surpreendentemente comum de as pistas individuais concordarem quanto ao desvio com predicado — como desviar com base em um parâmetro que é o mesmo para todas as pistas, de modo que os bits de máscara ativa sejam todos 0 ou todos 1 —, o desvio pula as instruções THEN ou as instruções ELSE.

Essa flexibilidade faz parecer que um elemento tem seu próprio contador de programa. Entretanto, no caso mais lento, somente uma lista SIMD poderia armazenar seu resultado

a cada dois ciclos de clock, com o restante ocioso. Um caso mais lento análogo para arquiteturas vetoriais é operar com um bit de máscara configurado para 1. Por causa dessa flexibilidade, programadores de GPU ingênuos podem apresentar um desempenho ruim, mas ela pode ser útil nos primeiros estágios do desenvolvimento de programas. Entretanto, tenha em mente que as únicas opções para uma pista SIMD em um ciclo de clock é realizar a operação especificada ou estar ocioso. Duas pistas SIMD não podem executar instruções diferentes ao mesmo tempo.

Tal flexibilidade também ajuda a explicar o nome *thread CUDA* dado a cada elemento em um thread de instruções SIMD, já que cria a ilusão de ação independente. Um programador inexperiente pode achar que essa abstração de thread significa que a GPU lida com desvios condicionais com mais graça. Alguns threads vão em uma direção, o restante vai em outra, o que parecerá ser verdadeiro enquanto você não estiver com pressa. Cada thread CUDA está executando a mesma instrução que qualquer outro thread no bloco de threads ou está ocioso. Essa sincronização torna mais fácil tratar loops com desvios condicionais, uma vez que o recurso de máscara pode desligar pistas SIMD e detectar automaticamente o fim do loop.

Às vezes, o desempenho resultante desmente essa simples abstração. Escrever programas que operam pistas SIMD nesse modo MIMD altamente independente é como escrever programas que usam grande parte do espaço de endereços virtuais em um computador com memória física menor. Os dois estão corretos, mas podem rodar tão lentamente que o programador pode ficar insatisfeito com o resultado.

Os compiladores vetoriais poderiam fazer os mesmos truques com registradores de máscara que as GPU fazem com hardware, mas isso envolveria instruções escalares para salvar, complementar e restaurar registradores de máscara. A execução condicional é um caso em que as GPUS fazem em tempo de execução de hardware o que as arquiteturas vetoriais fazem em tempo de compilação. Uma otimização disponível em tempo de execução para as GPUs, mas não em tempo de compilação para arquiteturas vetoriais, é pular as partes THEN ou ELSE quando os bits de máscara são todos 0 ou todos 1.

Assim, a eficiência com que as GPUs executam declarações condicionais se resume à frequência com que os desvios divergem. Por exemplo, um cálculo de autovalores tem aninhamento condicional profundo, mas medições do código mostram que cerca de 82% da emissão de ciclo de clock têm entre 29 e 32 dos 32 bits de máscara configurados para 1; então, as GPUs executam esse código com mais eficiência do que se poderia esperar.

Observe que o mesmo mecanismo trata o desdobramento de loops vetorial — quando o número de elementos não corresponde perfeitamente ao hardware. O exemplo do início desta seção mostra que uma declaração IF verifica se esse número de elementos de pista SIMD (armazenado em R8 no exemplo anterior) é menor do que o limite (`i < n`) e configura as máscaras de acordo com essa informação.

Estruturas de memória da GPU NVIDIA

A Figura 4.18 mostra as estruturas de memória de uma GPU NVIDIA. Cada pista SIMD em um processador SIMD multithreaded recebe uma seção privada de uma DRAM fora do chip, que chamamos *memória privada.* Ela é usada para a estrutura de pilha, para espalhamento de registradores e para variáveis privadas que não se encaixam nos registradores. As pistas SIMD não compartilham memórias privadas. As GPUS recentes armazenam temporariamente essa memória privada nas caches L1 e L2 para auxiliar o espalhamento de registradores e para acelerar as chamadas de função.

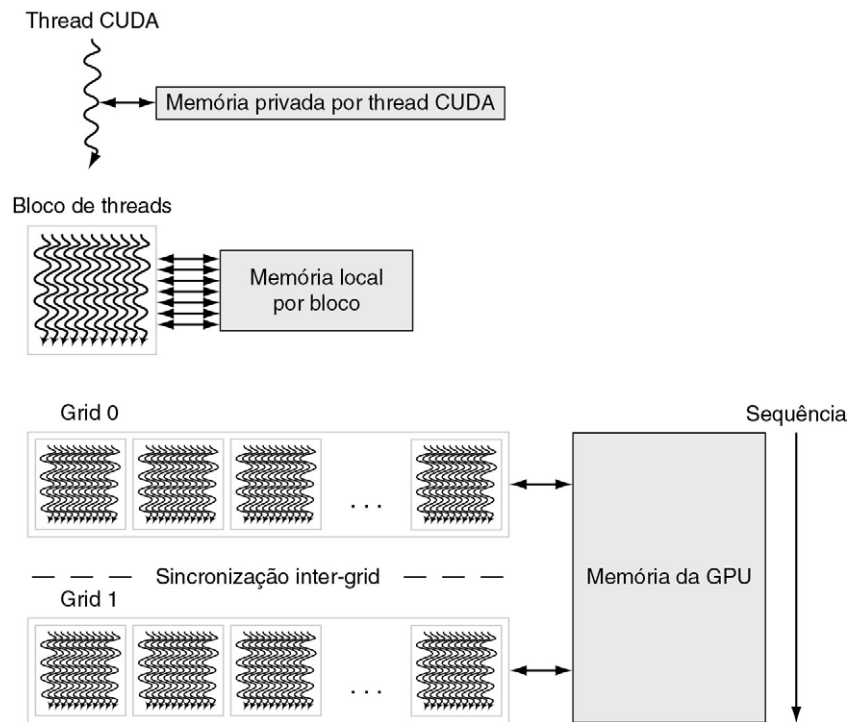



FIGURA 4.18 **Estruturas de memória de GPU.**
A memória de GPU é compartilhada por todos os grids (loops vetorizados), a memória local é compartilhada por todos os threads de instruções SIMD dentro de um bloco de threads (corpo de um loop vetorizado), e a memória privada é privativa de um único thread CUDA.

Chamamos *memória local* a memória no chip de cada processador SIMD multithreaded. Ela é compartilhada pelas pistas SIMD dentro de um processador SIMD multithreaded, mas não é compartilhada entre processadores SIMD multithread. O processador SIMD multithreaded aloca dinamicamente partes da memória local para um bloco de threads que cria o bloco de thread, liberando a memória quando todos os threads do bloco de threads saírem. Essa porção da memória local é privativa a esse bloco de thread.

Finalmente, chamamos *memória de GPU* a DRAM fora do chip compartilhada por toda a GPU e todos os blocos de threads. Nosso exemplo de multiplicação de vetores usou somente a memória da GPU.

O processador do sistema, chamado *host*, pode ler ou gravar na memória da GPU. A memória local não está disponível para o host, já que é privativa para cada processador SIMD multithread. Memórias privadas também não estão disponíveis para o host.

Em vez de depender de grandes caches para conter todos os conjuntos funcionais de uma aplicação, as GPUs tradicionalmente têm caches de streaming menores e dependem de intenso multithreading de threads de instruções SIMD para ocultar a grande latência da DRAM, uma vez que seus conjuntos funcionais podem ter centenas de megabytes. Dado o uso de multithreading para ocultar a latência de DRAM, em vez disso a área do chip usada para caches em processadores de sistema é gasta com recursos computacionais e no grande número de registradores para conter o estado de muitos threads de instruções SIMD. Em contraste, como mencionado, carregamentos e armazenamentos vetoriais amortizam a latência através de muitos elementos, já que eles somente pagam uma vez pela latência e então utilizam pipeline no restante dos acessos.

Embora ocultar a latência de memória seja a filosofia fundamental, observe que as GPUs e os processadores vetoriais mais recentes têm caches adicionais. Por exemplo, a recente arquitetura Fermi tem caches adicionais, mas elas são consideradas filtros de largura de banda para reduzir as demandas sobre a memória de GPU ou aceleradoras para as poucas variáveis cuja latência não pode ser ocultada por multithreading. Assim, a memória local para estruturas de pilha, chamadas de função e espalhamento de registradores é uma boa correspondência para as caches, já que a latência é importante quando se chama uma função. As caches também economizam energia, já que os acessos à cache no chip gastam muito menos energia do que acessos a múltiplos chips DRAM externos.

Para melhorar a largura de banda da memória e reduzir o overhead, como mencionado, as instruções de transferência de dados PTX reúnem requisições paralelas de thread do mesmo thread SIMD em uma única requisição de bloco de memória quando os endereços estão no mesmo bloco. Essas restrições são colocadas no programa da GPU, de modo análogo às orientações para programas do processador do sistema realizarem a pré-busca de hardware (Cap. 2). O controlador de memória da GPU também vai conter requisições e enviar requisições juntas para a mesma página aberta a fim de melhorar a largura de banda de memória (Seção 4.6). O Capítulo 2 descreve a DRAM em detalhes suficientes para que se compreendam os benefícios potenciais de agrupar endereços relacionados.

Inovações na arquitetura da GPU Fermi

O processador SIMD multithreaded da Fermi é mais complicado do que a versão simplificada na Figura 4.14. Para aumentar a utilização de hardware, cada processador SIMD tem dois escalonadores de thread SIMD e duas unidades de despacho de instruções. O escalonador de thread SIMD duplo seleciona dois threads de instruções SIMD e envia uma instrução de cada para dois conjuntos de 16 pistas SIMD, 16 unidades de carregamento/armazenamento ou quatro unidades de função especial. Assim, dois threads de instruções SIMD são escalonados a cada dois ciclos de clock para qualquer uma dessas coleções. Como os threads são independentes, não há necessidade de verificar se há dependências de dados no fluxo de instruções. Essa inovação é análoga a um processador vetorial multithreaded, que pode despachar instruções vetoriais de dois threads independentes.

A Figura 4.19 mostra o escalonador duplo enviando instruções, e a Figura 4.20 mostra o diagrama de blocos do processador SIMD multithreaded de uma GPU Fermi.

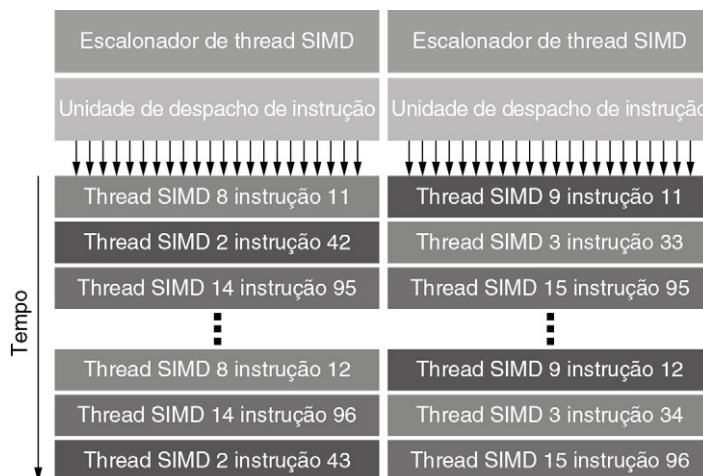



FIGURA 4.19 Diagrama de blocos de um escalonador de thread SIMD duplo.
Compare esse projeto ao projeto de thread SIMD único na Figura 4.16.

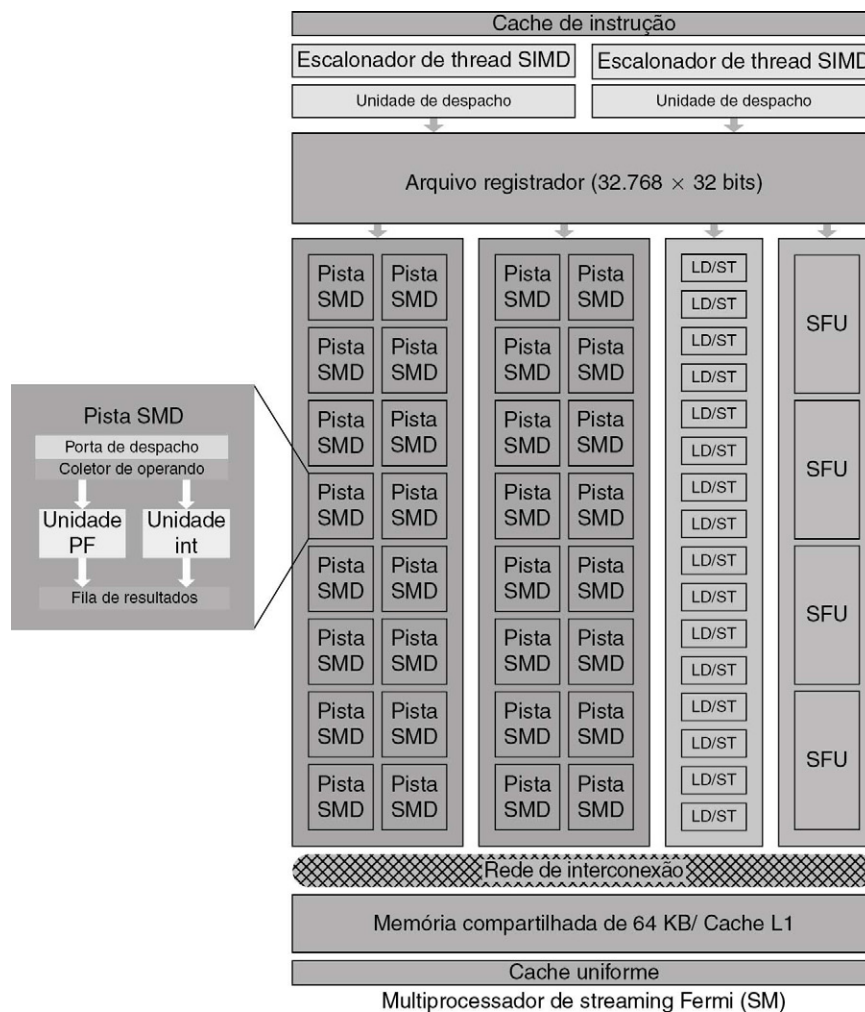



FIGURA 4.20 **Diagrama de blocos do processador SIMD multithreaded de uma GPU Fermi.**
Cada pista SIMD tem uma unidade de ponto flutuante pipelined, uma unidade de inteiros pipelined, alguma lógica para despacho de instruções e operandos para essas unidades, e uma fila para conter os resultados. As quatro unidades de função especial (SFUs) calculam funções como raízes quadradas, recíprocos, senos e cossenos.

A Fermi introduz diversas inovações para trazer as GPUs muito mais perto dos processadores de sistema mais populares do que a Tesla e as gerações anteriores de arquiteturas de GPU:

- *Aritmética rápida de ponto flutuante de precisão dupla.* A Fermi iguala a velocidade de precisão dupla dos processadores convencionais em cerca de metade da velocidade da precisão simples contra um décimo da velocidade da precisão simples na geração Tesla anterior. Ou seja, não há tentação, em qualquer ordem de magnitude, em usar a precisão simples quando a exatidão pede precisão dupla. O pico de desempenho de precisão dupla aumentou de 78 GFLOP/s na GPU anterior para 515 GFLOP/s com o uso de instruções multiplicação-soma.
- *Caches para memória de GPU.* Embora a filosofia da GPU seja ter threads suficientes para ocultar a latência da DRAM, existem variáveis que são necessárias para vários threads, como as variáveis locais mencionadas. A Fermi inclui uma cache de dados L1 e uma cache de instruções L1 para cada processador SIMD multithreaded e uma única cache L2 de 768 KB compartilhada por todos os processadores SIMD

multithreaded na GPU. Como mencionado, além de reduzir a pressão sobre a largura de banda da memória GPU, as caches podem economizar mais energia por estar no próprio chip do que as DRAM, que não estão no mesmo chip. A cache L1, na verdade, coabita a mesma SRAM da memória local. A arquitetura Fermi tem um bit, de modo que oferece a escolha de usar 64 KB de SRAM como uma cache L1 de 16 KB com 48 KB de memória local ou como uma cache L1 de 48 KB com 16 KB de memória local. Observe que o GTX 480 tem uma hierarquia de memória invertida: o tamanho do banco de registradores agregado é 2 MB, o tamanho de todas as caches L1 está entre 0,25-0,75 MB (dependendo de elas serem de 16 KB ou 48 KB), e o tamanho da cache L2 é de 0,75 MB. Será interessante ver o impacto dessa razão invertida sobre as aplicações de GPU.

- *Endereçamento de 64 bits e um espaço de endereços unificado para todas as memórias da GPU.* Essa inovação torna muito mais fácil fornecer os ponteiros necessários para C e C + +.
- *Códigos de correção de erro* (*ECC*) para detectar e corrigir erros na memória e nos registradores (Cap. 2). Para tornar aplicações de execução longa confiáveis em milhares de servidores, o ECC é uma norma nos centros de dados (Cap. 6).
- *Troca rápida de contexto.* Dado o grande estado de um processador SIMD multithreaded, a arquitetura Fermi tem suporte de hardware para trocar contextos muito mais rapidamente. Ela pode trocar em menos de 25 microssegundos, cerca de 10× mais rápido do que seu predecessor.
- *Instruções atômicas mais rápidas.* Incluídas pela primeira vez na arquitetura Tesla, a arquitetura Fermi melhora o desempenho das instruções atômicas em 5-20× , para poucos microssegundos. Uma unidade de hardware especial associada com a cache L2, e não situada dentro dos processadores SIMD multithreaded, lida com instruções atômicas.

Similaridades e diferenças entre arquiteturas vetoriais e GPUs

Como vimos, na verdade existem muitas similaridades entre arquiteturas vetoriais e GPUs. Além do jargão peculiar das GPUs, essas similaridades contribuíram para a confusão nos círculos arquitetônicos sobre quão novas as GPUs realmente são. Agora que você viu o que está por baixo dos capôs dos computadores vetoriais e das GPUs, pode apreciar tanto as similaridades quanto as diferenças. Já que as duas arquiteturas são projetadas para executar programas paralelos em nível de dados, mas tomam caminhos diferentes, essa comparação é feita em profundidade para termos melhor compreensão do que é necessário para o hardware DLP. A Figura 4.21 mostra primeiro o termo vetorial e depois o equivalente mais próximo em uma GPU.

Um processador SIMD é como um processador vetorial. Os múltiplos processadores SIMD nas GPUs agem como núcleos MIMD independentes, assim como muitos computadores vetoriais possuem múltiplos processadores vetoriais. Esse ponto de vista considera o NVIDIA GTX 480 uma máquina de 15 núcleos com suporte de hardware para multithreading, em que cada núcleo tem 16 pistas. A maior diferença é o multithreading, fundamental para as GPUs e ausente na maioria dos processadores vetoriais.

Examinando os registradores nas duas arquiteturas, o arquivo de registradores VMIPS contém vetores completos, ou seja, um bloco contíguo de 64 duplos. Em contraste, um único vetor em uma GPU seria distribuído através dos registradores de todas as pistas SIMD. Um processador VMIPS tem oito registradores vetoriais com 64 elementos, ou 512 elementos no total. Um thread GPU de instruções SIMD tem até 64 registradores com 32 elementos cada, ou 2.048 elementos. Esses registradores extras de GPU suportam multithreading.

Tipo	Termo vetorial	Termo de GPU CUDA/ NVIDIA mais próximo	Comentário
Abstrações de programa	Loop vetorizado	Grid	Os conceitos são similares, com a GPU usando o termo menos descritivo.
	Chime	--	Uma vez que uma instrução vetorial (instrução PTX) leva só dois ciclos no Fermi e quatro no Tesla para ser completada, um chime é curto nas GPUs.
Objetos de máquina	Instrução vetorial	Instrução PTX	Uma instrução PTX de um thread SIMD é transmitida para todas as pistas SIMD, então ela é similar a uma instrução vetorial.
	Gather/Scatter	Load/store global (ld.global/st.global)	Todos os carregamentos e armazenamentos em uma GPU são gather e scatter, no sentido em que cada pista SIMD envia um único endereço. Depende da unidade de fusão da GPU obter um desempenho de passo único quando os endereços das pistas SIMD o permitirem.
	Registradores de máscara	Registradores de predicado e registradores de máscara	Os registradores de máscara vetoriais são parte explícita do estado arquitetural, enquanto os registradores de máscara da GPU são internos ao hardware. O hardware condicional da GPU adiciona novos recursos além dos registradores de predicado para gerenciar dinamicamente as máscaras.
Hardware de processamento e memória	Processador vetorial	Processador SIMD multithread	Esses são similares, mas os processadores SIMD tendem a ter muitas pistas, levando alguns ciclos de clock por pista para completar um vetor, enquanto as arquiteturas vetoriais têm poucas pistas e levam muitos ciclos para completar um vetor. Eles também são multithreaded, enquanto os vetores normalmente não o são.
	Processador de controle	Escalonador de blocos de threads	O mais próximo é o escalonador de blocos de threads, que designa blocos de threads a um processador SIMD multithreaded. Mas as GPUs não possuem operações escalar-vetor nem instruções de transferência de dados por passos ou por passo unitário, o que os processadores de controle muitas vezes fornecem.
	Processador escalar	Processador de sistema	Devido à falta de memória compartilhada e à alta latência para a comunicação por um barramento PCI (1.000s de ciclos de clock), o processador de sistema em uma GPU raramente realiza as mesmas tarefas que um processador escalar realiza em um arquitetura vetorial.
	Pista vetorial	Pista SIMD	As duas são essencialmente unidades com registradores.
	Registradores vetoriais	Registradores de pista SIMD	O equivalente a um registrador vetorial é o mesmo registrador nas 32 pistas SIMD de um processador SIMD multithreaded executando um thread de instruções SIMD. O número de registradores por thread SIMD é flexível, mas o máximo é 64, então o número máximo de registradores vetoriais é 64.
	Memória principal	Memória da GPU	Memória para GPU *versus* memória de sistema no caso vetorial.

FIGURA 4.21 Equivalente de GPU para termos vetoriais.

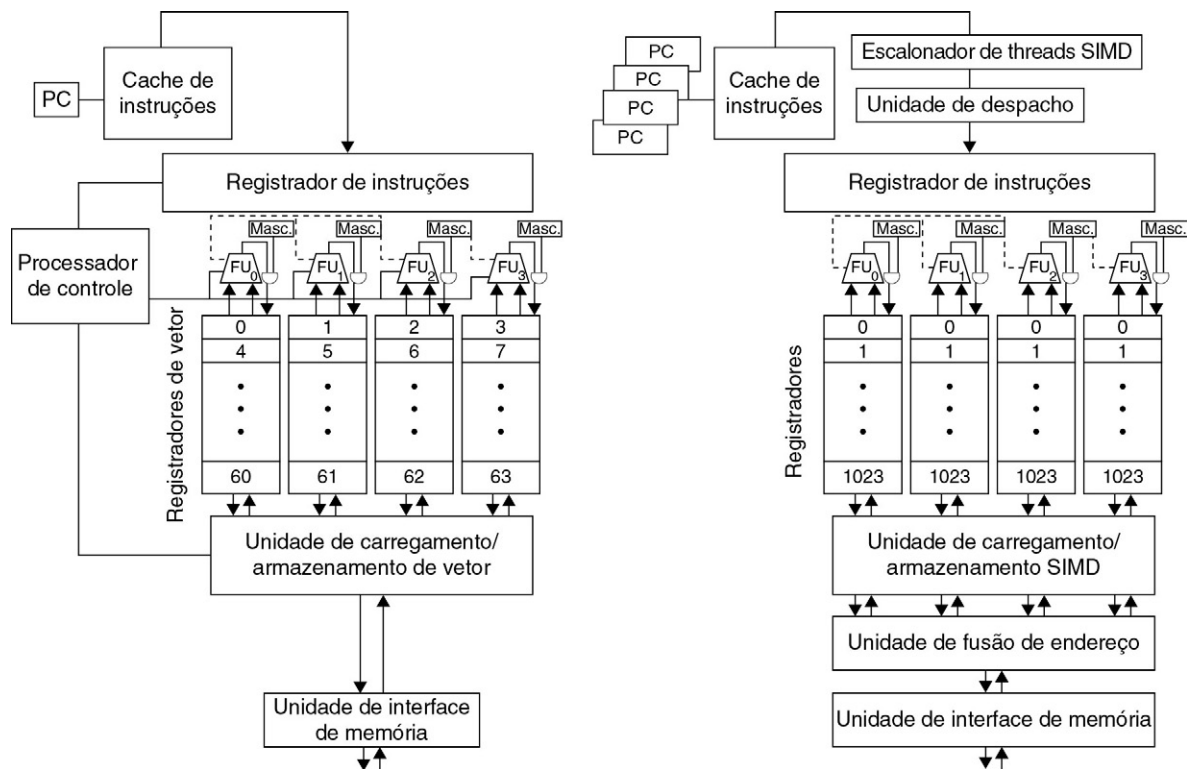



FIGURA 4.22 Processador vetorial com quatro pistas à esquerda e processador SIMD multithreaded de uma GPU com quatro pistas SIMD à direita. (As GPUs geralmente têm 8-16 pistas SIMD.)

O processador de controle fornece operandos escalares para operações escalar-vetor, incrementa o endereçamento para acessos por passo unitário e não unitário à memória e realiza outras operações do tipo contagem. O pico de desempenho de memória ocorre somente em uma GPU, quando a unidade de junção de endereços pode descobrir o endereçamento localizado. De modo similar, o pico de desempenho computacional ocorre quando todos os bits de máscara são configurados de modo idêntico. Observe que o processador SIMD tem um PC por thread SIMD para ajudar com o multithreading.

A Figura 4.22 é um diagrama de bloco das unidades de execução de um processador vetorial à esquerda e de um processador SIMD multithreaded de uma GPU à direita. Para fins pedagógicos, consideramos que o processador vetorial tem quatro pistas e o processador SIMD multithreaded também tem quatro pistas SIMD. Essa figura mostra que as quatro pistas SIMD agem de modo semelhante a uma unidade vetorial de quatro pistas e que um processador SIMD age de modo semelhante a um processador vetorial.

Na verdade, existem muito mais pistas nas GPUs, então os "chimes" de GPU são menores. Enquanto um processador vetorial pode ter 2-8 pistas e um comprimento vetorial de 32, por exemplo — gerando um chime de 4-16 ciclos —, um processador SIMD multithreaded pode ter oito ou 16 pistas. Um thread SIMD tem 32 elementos de largura, então um chime de GPU seria de somente dois ou quatro ciclos de clock. Essa diferença é o motivo de usarmos "processador SIMD" como o nome mais descritivo: ele é mais próximo de um projeto SIMD do que de um projeto tradicional de processador vetorial.

O termo de GPU mais próximo de um loop vetorizado é grid, e uma instrução PTX é o mais próximo de uma instrução vetorial, uma vez que um thread SIMD transmite uma instrução PTX para todas as pistas SIMD.

Com relação às instruções de acesso de memória nas duas arquiteturas, todos os carregamentos da GPU são instruções gather e todos os armazenamentos de GPU são instruções scatter. Se os endereços de dados dos threads CUDA se referirem a endereços

próximos que estão no mesmo bloco de cache/memória ao mesmo tempo, a unidade de junção de endereço da GPU vai garantir alta largura de banda de memória. As instruções de carregamento e armazenamento de passo unitário *explícito* das arquiteturas vetoriais *versus* o passo unitário *implícito* da programação de GPU são o motivo pelo qual escrever um código de GPU eficiente requer que os programadores pensem em termos de operações SIMD, embora o modelo de programação CUDA se pareça com MIMD. Como os threads CUDA geram seus próprios endereços, tanto com passo como com gather-scatter, os vetores de endereçamento são encontrados tanto nas arquiteturas vetoriais quanto nas GPUs.

Como mencionamos muitas vezes, as duas arquiteturas têm abordagens muito diferentes para ocultar a latência de memória. Arquiteturas vetoriais as amortizam para todos os elementos do vetor, tendo um acesso fortemente pipelined para que você pague a latência somente uma vez por carregamento ou armazenamento vetorial. Portanto, carregamentos e armazenamentos vetoriais são como uma transferência de bloco entre a memória e os registradores vetoriais. Em contraste, as GPUs ocultam a latência de memória usando multithreading. (Alguns pesquisadores estão investigando como adicionar multithreading às arquiteturas vetoriais para tentar capturar o melhor dos dois mundos.)

Com relação às instruções de desvio condicional, as duas arquiteturas as implementam usando registradores de máscara. Os dois caminhos de desvio condicional ocupam tempo e/ou espaço mesmo quando não armazenam um resultado. A diferença é que o compilador vetorial gerencia registradores de máscara explicitamente no software, enquanto o hardware e o assembler da GPU os gerenciam implicitamente usando marcadores de sincronização de desvio e uma pilha interna para salvar, complementar e restaurar máscaras.

Como mencionado, o mecanismo de desvio condicional das GPUS trata graciosamente o problema de desdobramento de loops das arquiteturas vetoriais. Quando o comprimento do vetor é desconhecido no momento da compilação, o programa deve calcular o módulo do comprimento do vetor de aplicação e o comprimento máximo do vetor, e armazená-lo no registrador de comprimento de vetor. Então, o loop expandido deve resetar o registrador de comprimento de vetor para o comprimento máximo do vetor pelo resto do loop. Esse caso é mais simples com as GPUs, uma vez que elas somente repetem o loop até que todas as pistas SIMD atinjam o limite do loop. Na última iteração, algumas pistas SIMD serão mascaradas e restauradas depois que o loop for completado.

O processador de controle de um computador vetorial tem um papel importante na execução de instruções vetoriais. Ele transmite operações para todas as pistas vetoriais e um valor de registrador escalar para operações vetor-escalar. Além disso, realiza cálculos implícitos que são explícitos nas GPUs, como incrementar automaticamente endereços de memória para carregamentos e armazenamentos de passo unitário e não unitário. A GPU não possui processador de controle. A analogia mais próxima é o escalonador de bloco de threads, que designa blocos de threads (corpos do loop do vetor) para processadores SIMD multithreaded. Os mecanismos de tempo de execução de hardware em uma GPU que gera endereços e então descobre se eles são adjacentes, o que é comum em muitas aplicações de DLP, provavelmente são menos eficientes em termos de energia do que um processador de controle.

O processador escalar em um computador vetorial executa as instruções escalares de um programa vetorial, ou seja, realiza operações que seriam muito lentas para serem feitas

na unidade vetorial. Embora o processador de sistema associado com uma GPU seja a analogia mais próxima para um processador escalar em uma arquitetura vetorial, os espaços de endereços separados e a transferência por um barramento PCIe significam milhares de ciclos de clock de overhead para usá-las em conjunto. O processador escalar pode ser mais lento do que um processador vetorial para cálculos de ponto flutuante em um computador vetorial, mas não na mesma razão de um processador do sistema *versus* um processador SIMD multithreaded (dado o overhead).

Portanto, cada "unidade vetorial" em uma GPU deve realizar cálculos que você esperaria realizar em um processador escalar em um computador vetorial. Ou seja, em vez de calcular no processador de sistema e comunicar os resultados, pode ser mais rápido desabilitar todas as pistas SIMD menos uma usando os registradores de predicado e máscaras embutidas, realizando o trabalho escalar com uma pista SIMD. É provável que o processador escalar em um computador vetorial, relativamente simples, seja mais rápido e mais eficiente em termos de energia do que a solução da GPU. Se os processadores de sistema e a GPU se tornarem mais intimamente ligados no futuro, será interessante ver se os processadores de sistema poderão ter o mesmo papel que os processadores escalares têm para as arquiteturas vetoriais e SIMD multimídia.

Similaridades e diferenças entre computadores SIMD multimídia e GPUs

Em um nível mais alto, os computadores multicore com extensões de instruções SIMD multimídia têm algumas similaridades com as GPUs. A Figura 4.23 resume as similaridades e diferenças.

Os dois são multiprocessadores cujos processadores usam múltiplas pistas SIMD, embora as GPUs tenham mais processadores e muitas outras pistas. Os dois usam multithreading de hardware para melhorar a utilização do processador, embora as GPUs tenham suporte de hardware para muitos outros threads. Inovações recentes nas GPUs significam que agora os dois têm taxas melhores de desempenho entre aritmética de ponto flutuante de precisão simples e precisão dupla. Os dois usam caches, embora as GPUs usem caches de streaming menores e os computadores multicore usem grandes caches multinível, que tentam conter completamente conjuntos funcionais inteiros. Os dois usam um espaço de endereços de 64 bits,

Recurso	Multicore com SIMD	GPU
Processadores SIMD	4 a 8	8 a 16
Pistas SIMD/processador	2 a 4	8 a 16
Suporte de hardware a multithreading para threads SIMD	2 a 4	16 a 32
Razão típica entre desempenho de precisão simples e precisão dupla	2:1	2:1
Maior tamanho de cache	8 MB	0,75 MB
Tamanho do endereço de memória	64-bit	64-bit
Tamanho da memória principal	8 GB a 256 GB	4 a 6 GB
Proteção de memória em nível de página	Sim	Sim
Paginação de demanda	Sim	Não
Processador escalar/processador SIMD integrado	Sim	Não
Coerência de cache	Sim	Não

FIGURA 4.23 Similaridades e diferenças entre multicore com extensões SIMD multimídia e GPUs recentes.

embora a memória física principal seja muito menor nas GPUs. As GPUs suportam proteção de memória em nível de página, mas não suportam paginação de demanda.

Além das grandes diferenças numéricas nos processadores, pistas SIMD, suporte de hardware a thread e tamanhos de cache, existem muitas diferenças entre as arquiteturas. As instruções de processador escalar e SIMD multimídia são fortemente integradas em computadores tradicionais. Elas são separadas por um barramento de E/S nas GPUs e têm memórias principais separadas. Os múltiplos processadores SIMD em uma GPU usam um único espaço de endereços, mas as caches não são coerentes como nos computadores multicore tradicionais. Ao contrário das GPUs, as instruções SIMD multimídia não suportam acessos gather-scatter à memória, o que a Seção 4.7 mostra ser uma omissão significativa.

Resumo

Agora que o véu foi levantado, podemos ver que, na verdade, as GPUs são somente processadores SIMD multithreaded, embora tenham mais processadores, mais pistas por processador e mais hardware de multithreading do que os computadores multicore tradicionais. Por exemplo, o Fermi GTX 480 tem 15 processadores SIMD com 16 pistas por processador e suporte de hardware a 32 threads SIMD. O Fermi até abraça o paralelismo em nível de instrução despachando instruções de dois threads SIMD para dois conjuntos de pistas SIMD. Eles também têm menos memória de cache — o cache L2 do Fermi é de 0,75 megabyte — e não são coerentes com o processador escalar distante.

O modelo de programação CUDA reúne todas essas formas de paralelismo ao redor de uma única abstração, o thread CUDA. Assim, o programador CUDA pode pensar em programar milhares de threads, embora na verdade eles estejam executando cada bloco de 32 threads nas muitas pistas dos muitos processadores SIMD. O programador CUDA que quer bom desempenho tem em mente que esses threads são bloqueados e executados 32 por vez e que os endereços precisam ser adjacentes para se obter bom desempenho do sistema de memória.

Embora tenhamos usado o CUDA e a GPU NVIDIA nesta seção, fique tranquilo, pois as mesmas ideias são encontradas na linguagem de programação OpenCL e em GPUs de outras empresas.

Agora que você entende melhor como as GPUs funcionam, vamos revelar o verdadeiro jargão. As Figuras 4.24 e 4.25 fazem a correspondência entre os termos descritivos e definições desta seção com os termos e definições oficiais da CUDA-NVIDIA e AMD. Nós incluímos também os termos OpenCL. Acreditamos que a curva de aprendizado de GPU seja acentuada em parte pelo fato de usarmos nomes como "multiprocessador de streaming" para o processador SIMD, "processador de thread" para a pista SIMD e "memória compartilhada" para a memória local — especialmente porque a memória local *não* é compartilhada entre processadores SIMD! Esperamos que essa abordagem em duas etapas faça você aumentar essa curva mais rapidamente, embora ela seja um pouco indireta.

4.5 DETECTANDO E MELHORANDO O PARALELISMO EM NÍVEL DE LOOP

Loops em programas são a origem de muitos dos tipos de paralelismo que discutimos aqui e veremos no Capítulo 5. Nesta seção, abordaremos a tecnologia de compilador para descobrir quanto paralelismo podemos explorar em um programa, além do suporte

Tipo	Nome mais descritivo usado neste livro	Termo oficial para GPU CUDA/NVIDIA	Definição do livro e termos AMD e OpenCL	Definição oficial CUDA/NVIDIA
Abstrações de programa	Loop vetorizável	Grid	Um loop vetorizável, executado na GPU, composto de um ou mais "blocos de thread" (ou corpos de loop vetorizado) que podem ser executados em paralelo. O nome do OpenCL é "faixa de índices". O nome AMD é "NDRange".	Um grid é um array de blocos de thread que podem ser executados ao mesmo tempo, sequencialmente, ou uma mistura dos dois.
	Corpo do loop vetorizável	Bloco de threads	Um loop vetorizado executado em um processador SIMD multithreaded, composto de um ou mais threads de instruções SIMD. Esses threads SIMD podem se comunicar através da memória local. O nome AMD e OpenCL é "grupo de trabalho".	Um bloco de threads é um array de threads CUDA executado ao mesmo tempo e pode cooperar e se comunicar através da memória compartilhada e da sincronização de barreira. Um bloco de threads tem um ID de bloco de threads dentro do seu grid.
	Sequência de operações de pista SIMD	Thread CUDA	Um corte vertical de um thread de instruções SIMD corresponde a um elemento executado por uma pista SIMD. O resultado é armazenado de acordo com a máscara. A AMD e a OpenCL chamam um thread CUDA "item de trabalho".	Um thread CUDA é um thread leve que executa um programa sequencial que pode cooperar com outros threads CUDA sendo executados no mesmo bloco de threads. Um thread CUDA tem um ID de thread dentro do seu bloco de threads.
Objeto de máquina	Um thread de SIMD	Warp	Um thread tradicional, mas contém somente instruções SIMD, que são executadas em um processador SIMD multithreaded. Os resultados são armazenados de acordo com uma máscara por elemento. O nome AMD é "frente de onda".	Um warp é um conjunto de threads CUDA paralelos (p. ex., 32) que executa a mesma instrução ao mesmo tempo em um processador SIMT/SIMD multithreaded.
	Instrução SIMD	Instrução PTX	Uma única instrução SIMD executada através das pistas SIMD. O nome AMD é instrução "AMDIL" ou "FSAIL".	Uma instrução PTX especifica uma instrução executada por um thread CUDA.

FIGURA 4.24 Conversão de termos usados neste capítulo para o jargão oficial NVIDIA/CUDA e AMD.
Os nomes OpenCL são dados na definição do livro.

de hardware para essas técnicas de compilador. Nós definimos precisamente quando um loop é paralelo (ou vetorizável), como a dependência pode impedir que um loop seja paralelo e técnicas para eliminar alguns tipos de dependência. Encontrar e manipular o paralelismo em nível de loop é essencial para explorar DLP e TLP, além das técnicas mais agressivas de ILP estático (p. ex., VLIW) que vamos examinar no Apêndice H.

O paralelismo em nível de loop normalmente é analisado no nível de fonte ou perto disso, enquanto a maior parte da análise do ILP é feita depois que as instruções são geradas pelo compilador. A análise em nível de loop envolve determinar quais dependências existem entre os operandos em um loop durante as iterações desse loop. Por enquanto, vamos considerar apenas as dependências de dados, que surgem quando um operando é escrito em algum ponto e lido em um ponto posterior. As dependências de nome também existem e podem ser removidas com técnicas de renomeação, como aquelas que exploramos no Capítulo 3.

A análise do paralelismo em nível de loop visa determinar se os acessos aos dados nas próximas iterações são dependentes dos valores de dados produzidos nas iterações anteriores; essa dependência é chamada *dependência transportada por loop*. A maioria dos exemplos

Tipo	Nome mais descritivo usado neste livro	Termo oficial para GPU CUDA/NVIDIA	Definição do livro e termos AMD e OpenCL	Definição oficial CUDA/NVIDIA
Hardware de processamento	Processador SIMD multithreaded	Multiprocessador de streaming	Processador SIMD multithreaded que executa um thread de instruções SIMD, independentemente de outros processadores SIMD. A AMD e o OpenCL o chamam de "unidade computacional". Entretanto, o programador CUDA escreve programas para uma pista em vez de para um "vetor" de múltiplas pistas SIMD.	Um multiprocessador de streaming (SM) é um processador SIMT/SIMD multithreaded que executa warps de threads CUDA. Um programa SIMT especifica a execução de um thread CUDA, em vez de um vetor de múltiplas pistas SIMD.
	Escalonador de bloco SIMD	Máquina Giga thread	Designa múltiplos corpos de loops vetorizados para processadores SIMD multithreaded. O nome AMD é máquina de despacho ultrathreaded".	Distribui e escalona blocos de threads de um grid para multiprocessadores de streaming conforme os recursos ficam disponíveis.
	Escalonador de thread SIMD	Escalonador de warp	Unidade de hardware que escalona e despacha threads de instruções SIMD quando elas estão prontas para serem executadas. Inclui um scoreboard para rastrear a execução de threads SIMD. O nome AMD é "escalonador de grupo de trabalho".	Um escalonador de warp em um multiprocessador de streaming escalona warps para execução quando sua próxima instrução está pronta para ser executada.
	Pista SIMD	Processador de thread	Pista SIMD de hardware que executa as operações em um thread de instruções SIMD em um único elemento. OS resultados são armazenados de acordo com a máscara. A OpenCL a chama "elemento de processamento". O nome AMD também é "pista SIMD".	Um processador de thread é um caminho de dados e porção de banco de registradores de um multiprocessador de streaming que executa operações para uma ou mais pistas de um warp.
Hardware de memória	Memória da GPU	Memória global	Memória DRAM acessível por todos os processadores SIMD multithreaded em uma GPU. A OpenCL a chama "memória global".	A memória global é acessível a todos os threads CUDA em qualquer bloco de threads em qualquer grid; implementada como região da DRAM e pode ser colocada em cache.
	Memória privada	Memória local	Parte da memória DRAM privada para cada pista SIMD. A AMD e a OpenCL as chamam "memória privada".	Memória "local do thread" privada de um thread CUDA; implementada como região em cache da DRAM.
	Memória local	Memória compartilhada	SRAM local rápida para um processador SIMD multithread, indisponível para outros processadores SIMD. A OpenCL a chama "memória local". A AMD a chama "memória de grupo".	Memória SRAM rápida compartilhada pelos threads CUDA compondo um bloco de threads e privada para aquele bloco de threads. Usada para comunicação entre threads CUDA em um bloco de threads nos pontos de sincronização de barreira.
	Registradores de pista SIMD	Registradores	Registradores em uma única pista SIMD alocada ao longo do corpo do loop vetorizado. A AMD também os chama "registradores".	Registradores privados para um thread CUDA; implementados como banco de registradores multithreaded para certas pistas de diversos warps para cada processador de thread.

FIGURA 4.25 **Conversão de termos usados neste capítulo para o jargão oficial NVIDIA/CUDA e AMD.**

Observe que nossa descrição utiliza os nomes "memória local" e "memória privada" usados na terminologia OpenCL. O NVIDIA utiliza SIMT, instruções únicas-múltiplos threads (single-instruction multiple-thread), em vez de SIMD, para descrever um multiprocessador de streaming. O SIMT é preferido no lugar do SIMD, e o fluxo de controle é diferente de qualquer máquina SIMD.

que consideramos nos Capítulos 2 e 3 não possui dependências transportadas por loop e, portanto, é paralela em nível de loop. Para ver se um loop é paralelo, vamos primeiro examinar a representação-fonte:

```
para (i=999; i>=0; i=i-1)
          x[i] = x[i] + s;
```

Nesse loop existe uma dependência entre os dois usos de x[i], mas ela está dentro de uma única iteração e não é transportada pelo loop. Existe uma dependência entre os usos sucessivos de *i* em iterações diferentes que é transportada pelo loop, mas ela envolve uma variável de indução e pode ser facilmente reconhecida e eliminada. Vimos exemplos de como eliminar dependências envolvendo variáveis de indução durante o desdobramento do loop na Seção 2.2 do Capítulo 2 e veremos exemplos adicionais mais adiante.

Como localizar o paralelismo em nível de loop envolve reconhecer estruturas como loops, referências de array e cálculos de variável de indução, o compilador pode fazer essa análise mais facilmente no nível de fonte ou quase isso, ao contrário do nível de código de máquina. Vejamos um exemplo mais complexo.

Exemplo Considere um loop como este:

```
para (i=0; i<100; i=i+1) {
      A[i+1] = A[i] + C[i];/*S1*/
      B[i+1] = B[i] + A[i+1]; /*S2*/
}
```

Suponha que *A*, *B* e *C* sejam arrays distintos, não sobrepostos. (Na prática, os arrays podem ser iguais ou se sobrepor. Como podem ser passados como parâmetros a um procedimento que inclui esse loop, determinar se os arrays se sobrepõem ou se são idênticos normalmente exige uma análise sofisticada entre os procedimentos do programa.) Quais são as dependências de dados entre as instruções S1 e S2 no loop?

Resposta Existem duas dependências diferentes:

1. S1 utiliza um valor calculado por S1 em uma iteração anterior, pois a iteração *i* calcula A[i + 1], que é lido na iteração *i* + 1. O mesmo acontece com S2 para B[i] e B[i + 1].
2. S2 usa o valor A[i + 1], calculado por S1 na mesma iteração.

Essas duas dependências são diferentes e possuem efeitos distintos. Para ver como elas diferem, vamos supor que haja somente uma dessas dependências de cada vez. Como a dependência da instrução S1 é sobre uma iteração anterior de S1, essa dependência é transportada pelo loop. Essa dependência força iterações sucessivas desse loop a serem executadas em série.

A segunda dependência (S2 dependendo de S1) ocorre dentro de uma iteração e não é transportada pelo loop. Assim, se essa fosse a única dependência, múltiplas iterações do loop poderiam ser executadas em paralelo, desde que cada par de instruções em uma iteração fosse mantido em ordem. Vimos esse tipo de dependência em um exemplo na Seção 2.2, em que o desdobramento foi capaz de expor o paralelismo. Essas dependências intraloop são comuns. Por exemplo, uma sequência de instruções vetoriais que usam encadeamento exibe claramente esse tipo de dependência.

Também é possível ter uma dependência transportada pelo loop que não impeça o paralelismo, como veremos no próximo exemplo.

Exemplo Considere um loop como este:

```
para (i=1; i<=100; i=i+1) {
      A[i] = A[i] + B[i]; /* S1 */
      B[i+1] = C[i] + D[i]; /* S2 */
}
```

Quais são as dependências entre S1 e S2? Esse loop é paralelo? Se não, mostre como torná-lo paralelo.

Resposta A instrução S1 usa o valor atribuído na iteração anterior pela instrução S2, de modo que existe uma dependência transportada pelo loop entre S2 e S1. Apesar dessa dependência, esse loop pode se tornar paralelo. Diferentemente do loop anterior, tal dependência não é circular: nenhuma instrução depende de si mesma e, embora S1 dependa de S2, S2 não depende de S1. Um loop será paralelo se puder ser escrito sem um ciclo nas dependências, pois a ausência de um ciclo significa que as dependências dão uma ordenação parcial nas instruções.
Embora não haja dependências circulares nesse loop, ele precisa ser transformado para estar de acordo com a ordenação parcial e expor o paralelismo. Duas observações são fundamentais para essa transformação:

1. Não existe dependência de S1 para S2. Se houvesse, haveria um ciclo nas dependências e o loop não seria paralelo. Como essa ordem de dependência é ausente, o intercâmbio das duas instruções não afetará a execução de S2.
2. Na primeira iteração do loop, a instrução S1 depende do valor de B[0] calculado *antes* do início do loop.

Essas duas observações nos permitem substituir o loop pela sequência de código a seguir:

```
A[0] = A[0] + B[0];
para (i=0; i<99; i=i+1) {
      B[i+1] = C[i] + D[i];
      A[i+1] = A[i+1] + B[i+1];
}
B[100] = C[99] + D[99];
```

A dependência entre as duas instruções não é mais transportada pelo loop, de modo que as iterações do loop podem ser sobrepostas desde que as instruções em cada iteração sejam mantidas em ordem.

Nossa análise precisa começar encontrando todas as dependências transportadas pelo loop. Essa informação de dependência é *inexata*, no sentido de que nos diz que tal dependência *pode* existir. Considere o exemplo a seguir:

```
para (i=1;i<100;i=i+1) {
      A[i] = B[i] + C[i]
      D[i] = A[i] * E[i]
}
```

A segunda referência a *A* nesse exemplo não precisa ser traduzida para uma instrução load, pois sabemos que o valor é calculado e armazenado pela instrução anterior; logo, a segunda referência a *A* pode ser simplesmente uma referência ao registrador no qual *A* foi calculado. Realizar essa otimização requer saber que as duas referências

são *sempre* para o mesmo endereço de memória e que não existe acesso intermediário ao mesmo local. Normalmente, a análise de dependência de dados só diz que uma referência *pode* depender de outra; é preciso haver uma análise mais complexa para determinar que duas referências *precisam ser* para o mesmo exato endereço. No exemplo anterior, uma versão simples dessa análise é suficiente, pois as duas referências estão no mesmo bloco básico.

Normalmente, as dependências transportadas pelo loop ocorrem na forma de uma *recorrência*. Uma recorrência ocorre quando uma variável é definida com base no valor dessa variável em uma iteração anterior, muitas vezes a imediatamente anterior, como no fragmento de código a seguir:

```
para (i=1;i<100;i=i+1) {
      Y[i] = Y[i-1] + Y[i];
}
```

Detectar uma recorrência pode ser importante por duas razões: algumas arquiteturas (especialmente as de computadores vetoriais) têm suporte especial para executar recorrências e, em um contexto ILP, talvez seja possível explorar uma boa quantidade de paralelismo.

Encontrando dependências

Obviamente, encontrar as dependências em um programa é importante tanto para determinar quais loops poderiam conter paralelismo quanto para eliminar dependências de nome. A complexidade da análise de dependência surge devido à presença de arrays e ponteiros em linguagens como C ou C + +, ou passagem de parâmetro por referência em FORTRAN. Como as referências de variável escalar se referem explicitamente a um nome, em geral podem ser facilmente analisadas com rapidez com o aliasing porque os ponteiros e os parâmetros de referência causam algumas complicações e incertezas na análise.

Como o compilador detecta dependências em geral? Quase todos os algoritmos de análise de dependência trabalham na suposição de que os índices de array são *afins*. Em termos mais simples, um índice de array unidimensional será afim se puder ser escrito na forma $a \times i + b$, onde a e b são constantes e i é a variável de índice do loop. O índice de um array multidimensional será afim se o índice em cada dimensão for afim. Os acessos a array disperso, que normalmente têm a forma x[y[i]], são um dos principais exemplos de acessos não afim.

Determinar se existe uma dependência entre duas referências para o mesmo array em um loop é, portanto, equivalente a determinar se duas funções afins podem ter o mesmo valor para diferentes índices entre os limites do loop. Por exemplo, suponha que tenhamos armazenado em elemento de array com valor de índice $a \times i + b$ e carregado do mesmo array com o valor de índice $c \times i + d$, onde i é a variável de índice do loop for que vai de m até n. Existirá uma dependência se duas condições forem atendidas:

1. Existem dois índices de iteração, j e k, ambos dentro dos limites do loop-for. Ou seja, $m \leq j \leq n$, $m \leq k \leq n$.
2. O loop armazena um elemento do array indexado por $a \times j + b$ e depois apanha esse *mesmo* elemento de array quando ele for indexado por $c \times k + d$. Ou seja, $a \times j + b = c \times k + d$.

Em geral, não podemos determinar se existe uma dependência em tempo de compilação. Por exemplo, os valores de a, b, c e d podem não ser conhecidos (podem ser valores em

outros arrays), o que torna impossível saber se existe uma dependência. Em outros casos, o teste de dependência pode ser muito dispendioso, mas decidido no momento da compilação. Por exemplo, os acessos podem depender dos índices de iteração de vários loops aninhados. Contudo, muitos programas contêm principalmente índices simples, em que *a*, *b*, *c* e *d* são constantes. Para esses casos, é possível criar alguns testes em tempo de compilação para a dependência.

Para exemplificar, um teste simples e satisfatório para a ausência de dependência é o teste do *maior divisor comum* (MDC). Ele é baseado na observação de que, se existir uma dependência transportada pelo loop, o MDC (c,a) precisa ser divisível por ($d - b$). (Lembre-se de que um inteiro, x, será *divisível* por outro inteiro, y, se obtivermos um quociente inteiro quando realizarmos a divisão y/x e não houver resto.)

Exemplo Use o teste do MDC para determinar se existem dependências no loop a seguir:

```
para (i=0; i<=100; i=i+1) {
      X[2*i+3] = X[2*i] * 5.0;
}
```

Resposta Dados os valores $a = 2$, $b = 3$, $c = 2$ e $d = 0$, então o MDC(a,c) = 2, e $d - b = -3$. Como 2 não é divisível por −3, nenhuma dependência é possível.

O teste do MDC é suficiente para garantir que não existe dependência; porém, existem casos em que o teste do MDC tem sucesso, mas não existe dependência. Isso pode surgir, por exemplo, porque o teste do MDC não leva em consideração os limites do loop.

Em geral, determinar se uma dependência realmente existe é incompleto. Na prática, porém, muitos casos comuns podem ser analisados com precisão a baixo custo. Recentemente, técnicas que usam uma hierarquia de testes exatos aumentando em generalidade e custos foram consideradas precisas e eficientes. (Um teste é *exato* se ele determina com precisão se existe uma dependência. Embora o caso geral seja incompleto, existem testes exatos para situações restritas que são muito mais baratos.)

Além de detectar a presença de uma dependência, um compilador deseja classificá-la quanto ao tipo. Essa classificação permite que um compilador reconheça as dependências de nome e as elimine durante a compilação, renomeando e copiando.

Exemplo O loop a seguir possui múltiplos tipos de dependência. Encontre todas as dependências verdadeiras, as dependências de saída e as antidependências, e elimine as dependências de saída e as antidependências pela renomeação.

```
para (i=0; i<=100; i=i+1) {
     Y[i] = X[i] / c; /* S1 */
     X[i] = X[i] + c; /* S2 */
     Z[i] = Y[i] + c; /* S3 */
     Y[i] = c - Y[i]; /* S4 */
 }
```

Resposta As dependências a seguir existem entre as quatro instruções:

1. Existem dependências verdadeiras de S1 para S3 e de S1 para S4, devido a Y[i]. Estas não são transportadas pelo loop, de modo que não impedem que o loop seja considerado paralelo. Essas dependências forçarão S3 e S4 a esperar que S1 termine.
2. Existe uma antidependência de S1 para S2, com base em X[i].
3. Existe uma antidependência de S3 para S4 para Y[i].
4. Existe uma dependência de saída de S1 para S4, com base em Y[i].

A versão do loop a seguir elimina essas falsas (ou pseudo) dependências.

```
para (i=0; i<=100; i=i+1 {
     T[i] = X[i] / c; /* Y renomeado para T para remover dependência de saída */
     X1[i] = X[i] + c;/* X renomeado para X1 para remover antidependência */
     Z[i] = T[i] + c;/* Y renomeado para T para remover antidependência */
     Y[i] = c - T[i];
 }
```

Após o loop, a variável *X* foi renomeada para X1. No código seguinte ao loop, o compilador pode simplesmente substituir o nome *X* por X1. Nesse caso, a renomeação não exige uma operação de cópia real, mas pode ser feita substituindo nomes ou pela alocação de registrador. Porém, em outros casos, a renomeação exigirá a cópia.

A análise de dependência é uma tecnologia essencial para explorar o paralelismo, assim como para o bloqueio similar da transformação abordada no Capítulo 2. A análise de dependência é a ferramenta básica para detectar o paralelismo em nível de loop. Compilar efetivamente os programas para computadores vetoriais, computadores SIMD ou multiprocessadores é algo que depende criticamente dessa análise. A principal desvantagem da análise de dependência é que ela só se aplica sob um conjunto limitado de circunstâncias, a saber, entre referências dentro de um único aninhamento de loop e usando funções de índice afins. Portanto, há uma grande variedade de situações em que a análise de dependência orientada a array *não pode* nos dizer o que poderíamos querer saber, por exemplo, analisar acessos realizados com ponteiros, e não com índices de array, pode ser muito mais difícil. (Essa é uma das razões pelas quais o FORTRAN é preferido ao C e C + + em muitas aplicações científicas projetadas para computadores paralelos.) Da mesma forma, analisar referências através de chamadas de procedimento é extremamente difícil. Assim, embora a análise de código escrito em linguagens sequenciais continue sendo importante, precisamos também de técnicas como OpenMP e CUDA, que escrevem loops explicitamente paralelos.

Eliminando cálculos dependentes

Como mencionado, uma das formas mais importantes de cálculo dependente é uma recorrência. Um produto de ponto é um exemplo perfeito de uma recorrência:

```
para (i=9999; i>=0; i=i-1)
      sum = sum + x[i] * y[i];
```

Esse loop não é recorrente, porque possui uma dependência carregada pelo loop na variável sum. Entretanto, podemos transformá-lo em um conjunto de loops: um deles é completamente paralelo e o outro pode ser parcialmente paralelo. O primeiro loop executará a parte completamente paralela desse loop. Ele é assim:

```
para (i=9999; i>=0; i=i-1)
      sum[i] = x[i] * y[i];
```

Observe que sum foi expandido de um escalar para um vetor (uma transformação chamada *expansão escalar*) e que essa transformação torna o novo loop completamente paralelo. Quando acabamos, entretanto, precisamos realizar o passo de redução, que soma os elementos do vetor. Ele é assim:

```
para (i=9999; i>=0; i=i-1)
      finalsum = finalsum + sum[i];
```

Embora esse loop não seja paralelo, tem uma estrutura bastante específica chamada *redução*. Reduções são comuns em Álgebra Linear e, como veremos no Capítulo 6, são também parte importante da primitiva de paralelismo primário MapReduce, usada em computadores em escala warehouse. Em geral, qualquer função pode ser usada como operador de redução, e casos comuns incluem operadores como max e min.

Às vezes, as reduções são tratadas por um hardware especial em um vetor e arquitetura SIMD, que permite que o passo de redução seja realizado muito mais rapidamente do que seria em modo escalar. Elas funcionam implementando uma técnica similar ao que pode ser feito em um ambiente de multiprocessador. Embora a transformação geral funcione com qualquer número de processadores, para simplificar suponha que tenhamos 10 processadores. No primeiro passo, que se refere a reduzir a soma, cada processador executa o seguinte (com *p* como o número de processador, indo de 0 a 9):

```
para (i=999; i>=0; i=i-1)
      finalsum[p] = finalsum[p] + sum[i+1000*p];
```

Esse loop, que soma 1.000 elementos em cada um dos 10 processadores, é completamente paralelo. Assim, um loop escalar simples pode completar o somatório das últimas 10 somas. Abordagens similares são usadas em processadores vetoriais e SIMD.

É importante observar que essa transformação depende da associatividade da adição. Embora a aritmética com alcance e precisão ilimitados seja associativa, a aritmética dos computadores não é associativa, seja aritmética de inteiros, pelo alcance limitado, seja aritmética de ponto flutuante, pelo alcance e pela precisão. Assim, às vezes usar essas técnicas de reestruturação pode levar a um comportamento errôneo, embora tais ocorrências sejam raras. Por esse motivo, a maioria dos compiladores requer que as otimizações que dependem da associatividade sejam habilitadas explicitamente.

4.6 QUESTÕES CRUZADAS

Energia e DLP: lento e largo *versus* rápido e estreito

Uma vantagem energética fundamental das arquiteturas paralelas em nível de dados vem da equação energética do Capítulo 1. Como consideramos amplo o paralelismo em nível de dados, o desempenho será o mesmo se reduzirmos a taxa de clock à metade e dobrarmos os recursos de execução: duas vezes o número de pistas para um computador vetorial, registradores e ALUs mais largos para SIMD multimídia e mais pistas SIMD para GPUs. Se pudermos reduzir a voltagem e, ao mesmo tempo, reduzir a taxa de clock, poderemos realmente reduzir a energia tanto quanto a potência para a computação, mantendo o mesmo pico de desempenho. Portanto, os processadores DLP tendem a ter taxas de clock

menores do que os processadores de sistema, cujo desempenho depende de altas taxas de clock (Seção 4.7).

Comparados aos processadores fora de ordem, os processos DLP podem ter lógica de controle mais simples para lançar um grande número de operações por ciclo de clock. Por exemplo, o controle é idêntico para todas as pistas em processadores vetoriais, e não existe lógica para decidir entre despacho múltiplo de instrução e lógica de execução especulativa. As arquiteturas vetoriais também podem tornar mais fácil desligar partes não utilizadas do chip. Cada instrução vetorial descreve explicitamente todos os recursos de que precisa para um número de ciclos quando a instrução é despachada.

Memória em banco e memória gráfica

A Seção 4.2 destacou a importância da largura de banda de memória significativa em arquiteturas vetoriais para suportar passo unitário, passo não unitário e acessos gather-scatter.

Para atingir alto desempenho, as GPUs também requerem largura de banda substancial de memória. Chips DRAM especiais projetados somente para GPUs, chamados GDRAM, de DRAM gráfica (*Graphics DRAM*), ajudam a fornecer essa largura de banda. Chips GDRAM têm maior largura de banda, muitas vezes com capacidade menor do que chips DRAM convencionais. Para fornecer essa largura de banda, muitas vezes os chips GDRAM são soldados diretamente na mesma placa da GPU, em vez de serem colocados em módulos DIMM, que são inseridos em slots em uma placa, como é o caso da memória de sistema. Os módulos DIMM permitem capacidade muito maior e atualização do sistema, ao contrário da GDRAM. Essa capacidade limitada — cerca de 4 GB em 2011 — está em conflito com o objetivo de executar problemas maiores, que é um uso natural da capacidade computacional maior das GPUs.

Para apresentar o melhor desempenho possível, as GPUs tentam levar em conta todos os recursos das GDRAMs. Em geral são organizados internamente em 4-8 bancos, com número de linhas sendo uma potência de 2 (tipicamente 16.384) e uma potência de 2 de bits por linha (tipicamente 8.192). O Capítulo 2 descreve os detalhes do comportamento da DRAM que as GPUs tentam igualar.

Dadas todas as demandas potenciais das tarefas de cálculo e de aceleração gráfica sobre as GDRAMs, o sistema de memória poderia deparar com grande número de requisições não correlacionadas. Infelizmente, essa diversidade prejudica o desempenho de memória. Para lidar com isso, o controlador de memória da GPU mantém listas separadas de tráfego limitadas para diferentes bancos de GDRAM, aguardando até que haja tráfego suficiente para justificar abrir uma linha e transferir todos os dados requisitados de uma vez. Esse atraso melhora a largura de banda, mas aumenta a latência, e o controlador deve garantir que nenhuma unidade de processamento fique com fome enquanto espera por dados, caso contrário, os processadores vizinhos poderão ficar ociosos. A Seção 4.7 mostra que as técnicas gather-scatter e as de acesso ciente dos bancos de memória podem apresentar aumentos substanciais no desempenho em comparação com as arquiteturas convencionais baseadas em cache.

Acessos por passo e perdas de TLB

Um problema com acessos por passo é como eles interagem com o buffer lookaside de tradução (TLB) para a memória virtual em arquiteturas vetoriais ou GPUs. (As GPUs usam as TLBs para mapeamento de memória.) Dependendo de como a TLB é organizada e do tamanho do array sendo acessado na memória, é possível até conseguir uma perda de TLB para cada acesso a um elemento no array!

	NVIDIA Tegra 2	NVIDIA Fermi GTX 480
Mercado	Cliente móvel	Desktop, servidor
Processador de sistema	Dual-Core ARM Cortex-A9	Não aplicável
Interface do sistema	Não aplicável	PCI Express 2.0 × 16
Largura de banda da interface do sistema	Não aplicável	6 GBytes/s (cada direção), 12 GBytes/s (total)
Taxa de clock	Até 1 GHz	1,4 GHz
Multiprocessadores SIMD	Indisponível	15
Pistas SIMD/multiprocessador SIMD	Indisponível	32
Interface de memória	32-bit LP-DDR2/DDR2	384-bit GDDR5
Largura de banda de memória	2,7 GBytes/s	177 GBytes/s
Capacidade de memória	1 GByte	1,5 GBytes
Transistores	242 M	3.030 M
Processo	40 nm TSMC processo G	40 nm TSMC processo G
Área do die	57 mm^2	520 mm^2
Potência	1,5 watt	167 watts

FIGURA 4.26 **Características principais das GPUs para clientes móveis e servidores.**
O Tegra 2 é a plataforma de referência para o OS Android e é encontrado no telefone celular LG Optimus 2X.

4.7 JUNTANDO TUDO: GPUs MÓVEIS *VERSUS* GPUs SERVIDOR TESLA *VERSUS* CORE i7

Dada a popularidade das aplicações gráficas, hoje as GPUs são encontradas em clientes móveis, além de servidores tradicionais ou computadores desktop para trabalho pesado. A Figura 4.26 lista as principais características do NVIDIA Tegra 2 para clientes móveis, que é usado no LG Optimus 2X e roda o SO Android, e a GPU Fermi para servidores. Os engenheiros de GPUs servidor esperam ser capazes de realizar animação ao vivo dentro de cinco anos depois de um filme ser lançado. Os engenheiros de GPUs móveis, por sua vez, querem que em mais cinco anos um cliente móvel possa fazer tudo o que um servidor ou console de games faz hoje. Mais concretamente, o objetivo geral é que a qualidade dos gráficos de um filme como *Avatar* seja atingida em tempo real em uma GPU servidor em 2015 e na sua GPU móvel em 2020.

O NVIDIA Tegra 2 para dispositivos móveis fornece tanto o processador de sistema quanto a GPU em um único chip usando uma única memória física. O processador de sistema é um ARM Cortex-A9 dual core, com cada núcleo usando execução fora de ordem e despacho duplo de instruções. Cada núcleo inclui a unidade de ponto flutuante opcional.

A GPU possui hardware para sombreamento programável de pixel, vértice e iluminação programáveis e gráficos 3-D, mas não inclui os recursos computacionais de GPU necessários para executar programas CUDA ou OpenCL.

O tamanho do die é de 57 mm^2 (7,5 × 7,5 mm) em um processo TSMC de 40 nm e contém 242 milhões de transistores. Ele usa 1,5 watt.

O NVIDIA 480, na Figura 4.26, é a primeira implementação da arquitetura Fermi. A taxa de clock é de 1,4 GHz e inclui 15 processadores SIMD. O próprio chip tem 16, mas, para melhorar o rendimento, somente 15 deles precisam funcionar para esse produto. O caminho para a memória GDDR5 tem 384 (6 × 64) bits de largura e interface com o clock a 1,84 GHz, oferecendo um pico de largura de banda de memória de 177 GBytes/s e

	Core i7-960	GTX 280	GTX 480	Razão 280/i7	Razão 480/i7
Número de elementos de processamento (núcleos ou SMs)	4	30	15	7,5	3,8
Frequência de clock (GHz)	3.2	1,3	1,4	0,41	0,44
Tamanho do die	263	576	520	2,2	2,0
Tecnologia	Intel 45 nm	TSMC 65 nm	TSMC 40 nm	1,6	1,0
Potência (chip, não módulo)	130	130	167	1,0	1,3
Transistores	700 M	1.400 M	3.030 M	2,0	4,4
Largura de banda de memória (GBytes/s)	32	141	177	4,4	5,5
Largura do SIMD de precisão, simples	4	8	32	2,0	8,0
Largura do SIMD de precisão dupla	2	1	16	0,5	8,0
Pico de FLOPs escalares de precisão simples (GFLOP/s)	26	117	63	4,6	2,5
Pico de FLOPs SIMD de precisão simples (GFLOP/s)	102	311 a 933	515 ou 1.344	3,0-9,1	6,6-13,1
(SP 1 soma ou multiplicação)	N.A.	(311)	(515)	(3,0)	(6,6)
(SP 1 multiplicações-somas de instrução fundida)	N.A.	(622)	(1.344)	(6,1)	(13,1)
(SP raro multiplicação-soma e multiplicação de despacho duplo fundido)	N.A.	(933)	N.A.	(9,1)	--
Pico de FLOPs SIMD de precisão dupla (GFLOP/s)	51	78	515	1,5	10,1

FIGURA 4.27 Especificações do Intel Core i7-960, NVIDIA GTX 280 e GTX 480.
As colunas à direita mostram as razões entre o GTX 280 e GTX 480 e Core i7. Para FLOPS SIMD de precisão simples no GTX 280, a velocidade mais alta (933) vem de um caso muito raro de despacho duplo de multiplicação-soma e multiplicação fundidos. O mais razoável é 622 para multiplicações-somas únicas fundidas. Embora o estudo de caso compare o 280 e o i7, incluímos o 480 para mostrar seu relacionamento com o 280, já que ele é descrito neste capítulo. Observe que essas larguras de banda de memória são maiores do que as apresentadas na Figura 4.28, porque são larguras de banda de pinos DRAM, e as apresentadas na Figura 4.28 são os processadores como medidos por um programa de benchmark. *(Da Tabela 2 em Lee* et al., *2010.)*

transferindo nas duas bordas do clock da memória de taxa dupla de dados. Ele se conecta com o processador do sistema host e a memória através de um link PCI Express 2.0 × 16, que tem um pico de taxa bidirecional de 12 GBytes/s.

Todas as características físicas do die GTX 480 são impressionantemente grandes: ele contém 30 bilhões de transistores, o tamanho do substrato é de 520 mm^2 (22,8 × 22,8 mm), em um processo TSMC de 40 nm, e a potência típica é de 167 watts. O módulo todo é de 250 watts, e inclui GPU, GDRAMs, ventoinhas, reguladores de potência etc.

Comparação entre uma GPU e uma MIMD com SIMD multimídia

Um grupo de pesquisadores da Intel publicou um artigo (Lee *et al.*, 2010) comparando um Intel i7 quadcore (Cap. 3) com extensões SIMD multimídia e GPU da geração anterior, a Tesla GTX 280. A Figura 4.27 lista as características dos dois sistemas. Ambos os produtos foram comprados no outono de 2009. O Core i7 usa a tecnologia semicondutora de 45 nm da Intel, enquanto a GPU usa a tecnologia de 65 nanômetros da TSMC. Embora possa ter sido mais justo fazer uma comparação com uma parte neutra ou as duas partes interessadas, o objetivo desta seção *não* é determinar o quanto um produto é mais rápido do que o outro, mas tentar entender o valor relativo dos recursos desses dois estilos arquitetônicos contrastantes.

Os rooflines do Core i7 920 e GTX 280, na Figura 4.28, ilustram as diferenças nos computadores. O 920 tem uma taxa de clock mais lenta do que o 960 (2,66 GHz *versus* 3,2 GHz), mas o resto do sistema é o mesmo. Não só o GTX tem largura de banda de memória

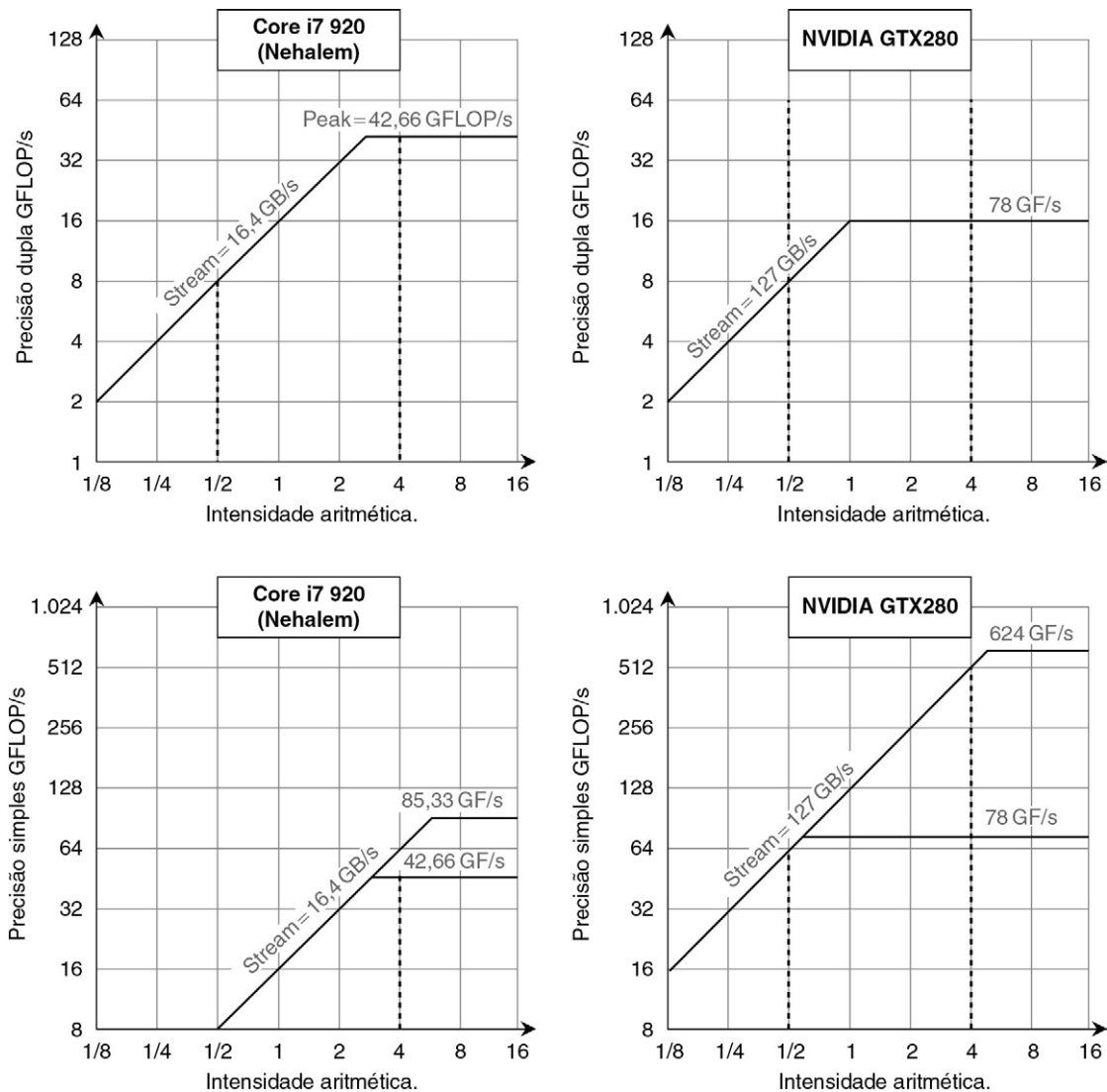



FIGURA 4.28 Modelo roofline (Williams *et al.*, 2009).

Esses rooflines mostram o desempenho de ponto flutuante de precisão dupla na linha superior e o desempenho de precisão simples na linha inferior. (O teto de desempenho de PF PD também está na linha inferior, para dar uma perspectiva.) O Core i7 920, à esquerda, tem um pico de desempenho PF PD de 42,66 GFLOP/s, um pico de PF PD de 85,33 GBytes/s e um pico de largura de banda de memória de 16,4 GBytes/s. O NVIDIA GTX 280 tem um pico de PF PD de 78 GFLOP/s, um pico de PF PS de 624 GBytes/s e um pico de largura de banda de memória de 127 GBytes/s. A linha vertical tracejada à esquerda representa uma intensidade aritmética de 0,5 FLOP/byte. Ela é limitada pela largura de banda de memória de não mais de 8 PD de GFLOP/s ou 8 PS GLOP/s no Core i7. A linha vertical tracejada à direita tem uma intensidade aritmética de 4 FLOP/byte. Ela é limitada somente computacionalmente a 42,66 PD GLOP/s e 64 PS GFLOP/s no Core i7 e 78 PD GLOP/s e 512 PD GFLOP/s no GTX 280. Para atingir a maior taxa computacional no Core i7 você precisa usar os quatro núcleos e instruções SSE com um número igual de multiplicações e somas. Para o GTX 280, você precisa usar instruções multiplicação-soma fundidas em todos os processadores SIMD multithreaded. Guz *et al.* (2009) têm um modelo analítico interessante para essas duas arquiteturas.

e desempenho de ponto flutuante de precisão dupla muito maiores, como seu ponto limite de precisão dupla também está consideravelmente à esquerda. Como mencionado, quanto mais à esquerda estiver o ponto limite do roofline, mais fácil será atingir o pico do desempenho computacional. O ponto limite de precisão dupla é 0,6 para o GTX 280 *versus* 2,6 para o Core i7. Para desempenho de precisão simples, o ponto limite se move para a direita, pois é muito mais difícil atingir o "telhado" do desempenho de precisão

Kernel	Aplicação	SIMD	TLP	Características
SGEMM (SGEMM)	Álgebra linear	Regular	Através de blocos 2-D	Calcula o limite após a colocação de blocos
Monte Carlo (MC)	Finanças computacionais	Regular	Através dos caminhos	Calcula limite
Convolução (Conv)	Análise de imagem	Regular	Através dos pixels	Calcula limite; limite BW para filtros pequenos
FFT (FFT)	Processamento de sinais	Regular	Através de FFTs	Calcula limite ou limite BW, dependendo do tamanho
SAXPY (SAXPY)	Produto escalar	Regular	Através do vetor	Limite BW para vetores grandes
LBM (LBM)	Migração por tempo	Regular	Através das células	Limite BW
Solucionador de restrição (Solv)	Físicas de corpo rígido	Gather/Scatter	Através das restrições	Limite de sincronização
SpMV (SpMV)	Solucionador esparso	Gather	Através de não zero	Limite BW para matrizes grandes típicas
GJK (GJK)	Detecção de colisão	Gather/Scatter	Através dos objetos	Calcula limite
Sort (Sort)	Base de dados	Gather/Scatter	Através de elementos	Calcula limite
Emissão de raios (RC)	Renderização de volume	Gather	Através de raios	Conjunto funcional de primeiro nível com 4-8 MB. Conjunto funcional de último nível acima de 500 MB
Busca (Serach)	Base de dados	Gather/Scatter	Através de pesquisas	Calcula o limite para árvore pequena; limite BW na base da árvore para árvore grande
Histograma (Hist)	Análise de imagem	Requer detecção de conflito	Através dos pixels	Limite de redução/sincronização

FIGURA 4.29 Características de throughput computacional de kernel (da Tabela 1 em Lee *et al.*, 2010).
Os nomes entre parênteses identificam o nome do benchmark nesta seção. Os autores sugerem que os códigos para as duas máquinas têm igual esforço de otimização.

simples, já que ele é muito mais alto. Observe que a intensidade aritmética do kernel se baseia nos bytes que vão para a memória principal, não nos que vão para a memória de cache. Assim, o uso da cache pode mudar a intensidade aritmética de um kernel em um computador em particular, considerando que a maioria das referências realmente vai para a cache. O roofline ajuda a explicar o desempenho relativo neste estudo de caso. Observe também que essa largura de banda é para acessos de passo unitário nas duas arquiteturas. Endereços gather-scatter reais que não são fundidos são mais lentos no GTX 280 e no Core i7, como podemos ver.

Os pesquisadores dizem que selecionaram os programas de benchmark analisando as características computacionais e de memória dos quatro sites recém-propostos de benchmark, "formulando o conjunto de *kernels de throughput computacional* que capturam essas características". A Figura 4.29 descreve esses 14 kernels, e a Figura 4.30 mostra os resultados de desempenho com números maiores significando maior velocidade.

Dado que as especificações de desempenho bruto do GTX 280 variam de 2,5× mais lento (taxa de clock) a 7,5× mais rápido (núcleos por chip), enquanto o desempenho varia de 2,0× mais lento (Solv) a 15,2× mais rápido (GJK), os pesquisadores da Intel exploraram as razões para essas diferenças:

- *Largura de banda de memória.* A GPU tem 4,4× a largura de banda de memória, o que ajuda a explicar por que o LBM e o SAXPY rodam 5,0× e 5,3× mais rápido, respectivamente. Seus conjuntos funcionais têm centenas de megabytes e, portanto, não se encaixam no cache do Core i7. (Para acessar intensamente a memória,

Kernel	Unidades	Core i7-960	GTX 280	GTX 280/i7-960
SGEMM	GFLOP/s	94	364	3,9
MC	Bilhões de caminhos/s	0,8	1.4	1,8
Conv	Milhões de pixels/s	1.250	3.500	2,8
FFT	GFLOP/s	71,4	213	3,0
SAXPY	GBytes/s	16,8	88,8	5,3
LBM	Milhões de buscas/s	85	426	5,0
Solv	Quadros/s	103	52	0,5
SpMV	GFLOP/s	4,9	9,1	1,9
GJK	Quadros/s	67	1.020	15,2
Sort	Milhões de elementos/s	250	198	0,8
RC	Quadros/s	5	8,1	1,6
Search	Milhões de buscas/s	50	90	1,8
Hist	Milhões de pixels/s	1.517	2.583	1,7
Bilat	Milhões de pixels/s	83	475	5,7

FIGURA 4.30 **Desempenhos brutos e relativos medidos para as duas plataformas.**
Nesse estúdio, o SAXPY é usado somente como medida de largura de banda de memória, então a unidade correta é GBytes/s, e não GFLOP/s. *(Baseado na Tabela 3 em Lee* et al., *2010.)*

eles não usam bloqueio de cache no SAXPY.) Assim, a rampa dos rooflines explica seu desempenho. O SpMV tem também um grande conjunto funcional, mas roda a somente 1,9 ×, porque o ponto flutuante de precisão dupla da GTX 280 é somente 1,5× mais rápida do que o Core i7. (Lembre-se de que a precisão dupla da Fermi GTX 480 é 4× mais rápida do que a Tesla GTX 280.)

- *Largura de banda computacional.* Cinco dos kernels restantes são limitados computacionalmente: SGEMM, Conv, FFT, MC e Bilat. A GTX é mais rápida 3,9, 2,8, 3,0, 1,8 e 5,7, respectivamente. Os três primeiros usam aritmética de ponto flutuante de precisão simples, e a precisão simples da GTX 280 é 3× a 6× mais rápida. (O "9× mais rápido que o Core i7", como mostrado na Figura 4.27, ocorre somente em um caso muito especial em que a GTX 280 pode despachar um multiplicação-soma fundida e uma multiplicação por ciclo de clock.) O MC usa precisão dupla, o que explica por que ele é somente 1,8× mais rápido, já que o desempenho de PD é somente 1,5× mais rápido. Bilat usa funções transcendentais, que a GTX 280 suporta diretamente (Fig. 4.17). O Core i7 passa dois terços do seu tempo calculando funções transcendentais, então a GTX 280 é 5,7× mais rápida. Essa observação ajuda a destacar o valor do suporte de hardware para operações que ocorrem na carga de trabalho: ponto flutuante de precisão dupla e, talvez, até mesmo transcendentais.
- *Benefícios de cache.* A emissão de raios (RC) é somente 1,6× mais rápida na GTX porque o bloqueio de caches nas caches do Core i7 o impede de se tornar limitado pela largura de banda de memória, como as GPUs. O bloqueio de cache pode ajudar também na busca. Se as árvores de índices forem pequenas de modo que se encaixem na cache, o Core i7 é duas vezes mais rápido. Árvores maiores de índices as tornam limitadas pela largura de banda de memória. No geral, a GTX 280 executa buscas 1,8× mais rapidamente. O bloqueio de cache também auxilia na ordenação (sort). Enquanto a maioria dos programadores não executaria a ordenação em um processador SIMD, ela pode ser escrita com uma primitiva de ordenação de 1 bit chamada *split*. Entretanto, o algoritmo split executa muito mais instruções do que

uma organização escalar. Como resultado, a GTX roda somente 0,8× mais rápido do que o Core i7. Observe que as caches também ajudam outros kernels no Core i7, já que o bloqueio de cache permite que SGEMM, FFT e SpMV se tornem limitados computacionalmente. Essa observação reenfatiza a importância das otimizações de bloqueio de cache no Capítulo 2. (Seria interessante ver como as caches da Fermi GTX 480 vão afetar os seis kernels mencionados neste parágrafo.)

- *Gather/Scatter.* As extensões SIMD multimídia serão de pouca ajuda se os dados estiverem espalhados pela memória principal. O desempenho ótimo vem somente quando os dados estão alinhados em limites de 16 bytes. Assim, GJK obtém pouco benefício do SIMD no Core i7. Como mencionado, as GPUs oferecem endereçamento gather-scatter, que é encontrado em uma arquitetura vetorial mas omitido nas extensões SIMD. A unidade de união de endereço também ajuda combinando acessos à mesma linha DRAM, reduzindo assim o número de gathers e scatters. O controlador de memória também reúne acessos à mesma página DRAM. Essa combinação significa que a GTX 280 executa o GJK 15,2× mais rápido do que o Core i7, que é mais do que qualquer parâmetro físico na Figura 4.27. Essa observação reforça a importância do gather-scatter para arquiteturas vetoriais e GPU, que está ausente nas extensões SIMD.
- *Sincronização.* A sincronização de desempenho é limitada pelas atualizações atômicas, que são responsáveis por 28% do tempo de execução total no Core i7, apesar de ele ter uma instrução busca e incremento de hardware. Assim, Hist é somente 1,7× mais rápido na GTX 280. Como mencionado, as atualizações atômicas da Fermi GTX 480 são 5× a 20× mais rápidas do que aquelas na Tesla GTX 280, então novamente seria interessante executar Hist na GPU mais nova. Solv soluciona um conjunto de restrições independentes com pouca computação, seguida por uma sincronização de barreira. O Core i7 se beneficia das instruções atômicas e um modelo de consistência de memória que garante os resultados corretos mesmo que nem todos os acessos à hierarquia de memória tenham sido completados. Sem o modelo de consistência de memória, a versão da GTX 280 lança alguns conjuntos do processador de sistema, o que leva a GTX 280 a rodar 0,5× mais rápido do que o Core i7. Essa observação destaca como o desempenho de sincronização pode ser importante para alguns problemas de dados paralelos.

É surpreendente com que frequência os pontos fracos na Tesla GTX 280 que foram descobertos por kernels selecionados pelos pesquisadores da Intel já haviam sido endereçados na arquitetura que sucedeu a Tesla. A Fermi tem desempenho de ponto flutuante de precisão dupla mais rápida, operações atômicas e caches. Em um estudo relacionado, pesquisadores da IBM fizeram a mesma observação (Bordawekar, 2010). É interessante também que o suporte gather-scatter das arquiteturas vetoriais, décadas mais antigo que as instruções SIMD, era tão importante para as eficazes utilidades dessas extensões SIMD que algumas pessoas haviam feito uma previsão antes dessa comparação (Gebis e Patterson, 2007). Os pesquisadores da Intel observaram que seis dos 14 kernels explorariam melhor o SIMD com suporte mais eficiente a gather-scatter no Core i7. Esse estudo certamente estabelece também a importância do bloqueio de cache. Será interessante ver se gerações futuras do hardware, compiladores e bibliotecas multicore e de GPU respondem com recursos que melhoram o desempenho de tais kernels.

Esperamos que haja mais dessas comparações multicore/GPU. Observe que um importante recurso ausente dessa comparação foi descrever o nível de esforço para obter os resultados para os dois sistemas. Idealmente, as comparações futuras liberariam o código usado nos dois sistemas para que outros pudessem recriar os mesmos experimentos em plataformas de hardware diferentes e, possivelmente, melhorar os resultados.

4.8 FALÁCIAS E ARMADILHAS

Enquanto paralelismo em nível de dados é a forma mais fácil de paralelismo após ILP da perspectiva do programador, e plausível a facilidade do ponto de vista dos arquitetos, ainda tem muitas falácias e armadilhas.

Falácia. *As GPUs sofrem por serem coprocessadores.*

Embora a divisão entre a memória principal e a memória de GPU apresente desvantagens, existem vantagens em estar distante da CPU.

Por exemplo, as PTX existem em parte por causa da natureza de dispositivo de E/S das GPUs. Esse nível de indireção entre o compilador e o hardware dá aos arquitetos de GPU muito mais flexibilidade do que os arquitetos de sistema têm. Muitas vezes é difícil saber com antecedência se uma inovação de arquitetura será bem suportada pelos compiladores e bibliotecas e se será importante para as aplicações. Às vezes, um novo mecanismo vai se mostrar útil para uma ou duas gerações e então decair em importância, conforme o mundo de TI mudar. As PTX permitem aos arquitetos de GPU tentar inovações especulativamente e abandoná-las em gerações subsequentes se elas forem um desapontamento ou perderem importância, o que encoraja a experimentação. A justificativa para a inclusão é compreensivelmente muito maior para processadores de sistema — e, portanto, muito menos experimentação pode ocorrer, já que distribuir código binário de máquina normalmente implica que novos recursos devem ser suportados por todas as gerações futuras daquela arquitetura.

Uma demonstração do valor das PTX é que a arquitetura Fermi mudou radicalmente o conjunto de instruções de hardware — de orientado à memória, como o ×86, para orientado a registrados, como no MIPS, *além* de dobrar o tamanho de endereço para 64 bits sem perturbar a pilha de software NVIDIA.

Armadilha. *Concentrar-se no pico de desempenho em arquiteturas vetoriais e ignorar o overhead de inicialização.*

Os primeiros processadores vetoriais memória-memória, como o TI ASC e o CDC STAR-100, têm longos tempos de inicialização. Para alguns problemas vetoriais, os vetores devem ter mais de 100 elementos para que o código vetorial seja mais rápido do que o código escalar! No CYBER 205 — derivado do STAR 100 —, o overhead de inicialização para o DAZPY é de 158 ciclos de clock, o que aumenta substancialmente o ponto de break-even. Se as taxas de clock do Cray-1 e do CYBER 205 fossem idênticas, o Cray-1 seria mais rápido até que o número de elementos no vetor fosse maior do que 64. Já que o clock do Cray-1 era também mais rápido (embora o 205 fosse mais novo), o ponto de cruzamento era um comprimento de vetor de mais de 100.

Armadilha. *Aumentar o desempenho vetorial sem aumentos comparáveis em desempenho escalar.*

Esse desequilíbrio foi um problema em muitos dos primeiros processadores vetoriais, e um ponto em que Seymour Cray (o arquiteto dos computadores Cray) reescreveu as regras. Muitos dos primeiros processadores vetoriais tinham unidades escalares comparativamente pequenas (além de grandes overheads de inicialização). Mesmo hoje, um processador com desempenho vetorial menor mas com desempenho escalar melhor pode ter melhor desempenho do que um processador com maior pico de desempenho vetorial. Um bom desempenho escalar mantém os custos de overhead baixos (p. ex., strip mining) e reduz o impacto da lei de Amdahl.

Um bom exemplo disso vem na comparação de um processador escalar rápido com um processador vetorial com desempenho escalar menor. Os kernels Livermore Fortran são uma coleção de 24 kernels científicos com graus variáveis de vetorização. A Figura 4.31

Processador	Taxa mínima para qualquer loop (MFLOPS)	Taxa máxima para qualquer loop (MFLOPS)	Média harmônica para todos os 24 loops (MFLOPS)
MIPS M/120-5	0,80	3,89	1,85
Stardent-1500	0,41	10,08	1,72

FIGURA 4.31 **Medidas de desempenho para os kernels Livermore Fortran em dois processadores diferentes.**
O MIPS M/120-5 e o Stardent-1500 (anteriormente o Ardent Titan-1) usam um chip MIPS R2000 de 16,7 MHz como CPU principal. O Stardent-1500 usa sua unidade vetorial para PF escalar e tem cerca de metade do desempenho escalar (como medido pela taxa mínima) do MIPS M/120-5, que usa o chip MIPS R2010 FP. O processador vetorial é mais de 2,5× mais rápido para um loop altamente vetorizável (taxa máxima). Entretanto, o desempenho escalar menor do Stardent-1500 nega o desempenho maior vetorial quando o desempenho total é medido pela média harmônica em todos os 24 loops.

mostra o desempenho de dois processadores diferentes nesse benchmark. Apesar do maior pico de desempenho do processador vetorial, seu baixo desempenho escalar o torna mais lento do que um processador escalar veloz, como medido pela média harmônica.

Hoje, o outro lado desse perigo é aumentar o desempenho vetorial 0, por exemplo, aumentando o número de pistas sem aumentar o desempenho escalar. Tal miopia é outro caminho para um computador desequilibrado.

A próxima falácia se relaciona intimamente com esta.

Falácia. *Você pode obter um bom desempenho vetorial sem fornecer largura de banda de memória.*

Como vimos no loop DAXPY e no modelo de Roofline, a largura de memória é muito importante para todas as arquiteturas SIMD. O DAXPY requer 1,5 referência de memória por operação de ponto flutuante, e essa taxa é típica para muitos códigos científicos. Mesmo se as operações de ponto flutuante não ocupassem tempo, um Cray-1 não poderia aumentar o desempenho da sequência vetorial usada, já que ela é limitada pela memória. O desempenho do Cray-1 em Linpack saltou quando o compilador usou bloqueio para mudar o cálculo de modo que os valores fossem mantidos nos registradores vetoriais. Essa abordagem diminuiu o número de referências de memória por FLOP e melhorou o desempenho por um fator de quase duas vezes! Assim, a largura de banda de memória no Cray-1 tornou-se suficiente para um loop que antes requeria mais largura de banda.

Falácia. *Nas GPUs, simplesmente adicione mais threads se não tiver desempenho de memória suficiente.*

As GPUs usam muitos threads CUDA para ocultar a latência da memória principal. Se os acessos de memória estiverem espalhados, mas não correlacionados entre threads CUDA, o sistema de memória vai ficar progressivamente mais lento em responder a cada requisição individual. Eventualmente, nem mesmo vários threads vão cobrir a latência. Para que a estratégia "mais threads CUDA" funcione, não só você precisará de muitos threads CUDA, como os próprios threads CUDA deverão se comportar bem em termos de localidade de acessos de memória.

4.9 CONSIDERAÇÕES FINAIS

O paralelismo em nível de dados está aumentando em importância para dispositivos pessoais móveis, dada a popularidade de aplicações mostrando a importância de áudio, vídeo e jogos nesses dispositivos. Quando combinados com um modelo mais fácil de programar do que o paralelismo em nível de tarefa e eficiência energética potencialmente melhor, é

fácil prever uma renascença do paralelismo em nível de dados na próxima década. De fato, já podemos ver essa ênfase em produtos, já que tanto as GPUs quanto os processadores tradicionais vêm aumentando o número de pistas SIMD tão rapidamente quanto estão adicionando processadores (Fig. 4.1 na página 229).

Portanto, estamos vendo os processadores de sistema adquirirem mais das características da GPUs e vice-versa. Uma das maiores diferenças entre o desempenho dos processadores convencionais e as GPUs tem sido o endereçamento gather-scatter. Arquiteturas vetoriais tradicionais mostram como adicionar tal endereçamento a instruções SIMD, e esperamos ver mais ideias das comprovadas arquiteturas vetoriais adicionadas às extensões SIMD ao longo do tempo.

Como dissemos nas páginas iniciais da Seção 4.4, a questão das GPUs não é simplesmente o fato de a arquitetura ser melhor, mas, dado o investimento em hardware para realizar gráficos bem, como ela pode ser melhorada para suportar computação mais geral. Embora no papel as arquiteturas vetoriais tenham muitas vantagens, ainda precisa ser provado que as arquiteturas vetoriais podem ser uma base tão boa para gráficos quanto as GPUS.

Os processadores SIMD das GPUs e computadores ainda têm um projeto relativamente simples. Técnicas mais agressivas provavelmente serão introduzidas ao longo do tempo para melhorar a utilização das GPUs, já que as aplicações de GPU para cálculo estão começando a ser desenvolvidas. Ao estudar esses novos programas, os projetistas de GPUs certamente vão descobrir e implementar novas otimizações de máquina. Uma questão é se os processadores escalares (ou processadores de controle), que servem para poupar hardware e energia em processadores vetoriais, vão aparecer dentro das GPUs.

A arquitetura Fermi já incluía muitos recursos encontrados em processadores convencionais para tornar as GPUs mais populares, mas ainda há outras necessárias para fechar a lacuna. Aqui estão algumas que esperamos ver endereçadas em futuro próximo.

- *GPUs virtualizáveis.* A virtualização se provou importante para os servidores e é a base da computação em nuvem (Cap. 6). Para que as GPUs sejam incluídas na nuvem, elas vão precisar ser tão virtualizáveis quanto os processadores e memórias a que estão ligadas.
- *Tamanho relativamente pequeno da memória das GPUs*. Um uso comum de computação mais rápida é solucionar problemas maiores, e problemas maiores muitas vezes têm uma "pegada" maior na memória. A inconsistência das GPUs entre velocidade e tamanho pode ser endereçada com mais capacidade de memória. O desafio é manter uma grande largura de banda e, ao mesmo tempo, aumentar a capacidade.
- *E/S diretas para a memória da GPU.* Programas reais usam E/S para dispositivos de armazenamento, assim como para buffers de quadro, e programas grandes podem exigir muitas E/S, além de uma memória de tamanho considerável. Os sistemas de GPU atuais devem transferir entre dispositivos de E/S e a memória de sistema e então entre a memória de sistema e a memória da GPU. Esse passo extra diminui significativamente o desempenho de E/S em alguns programas, tornando as GPUs menos atrativas. A lei de Amdahl nos alerta do que acontece quando se negligencia uma parte da tarefa enquanto se aceleram outras. Esperamos que as GPUs futuras tornem todos cidadãos de primeira classe de E/S, assim como fazem hoje para E/S com frame buffer.
- *Memórias físicas unificadas.* Uma solução alternativa para os dois itens anteriores é ter uma única memória física para o sistema e a GPU, assim como algumas GPUs baratas fazem para PMDs e laptops. A arquitetura AMD Fusion, anunciada quando

esta edição estava sendo finalizada, é uma fusão inicial entre as GPUs tradicionais e as CPUs tradicionais. A NVIDIA também anunciou o Projeto Denver, que combina um processador escalar ARM com GPUs NVIDIA em um único espaço de endereços. Quando esses sistemas forem comercializados, será interessante aprender quão integrados eles são, além do impacto da integração no desempenho e na energia de aplicações com paralelismo de dados e gráficos.

Tendo coberto as muitas versões de SIMD, o Capítulo 5 entrará no reino do MIMD.

4.10 PERSPECTIVAS HISTÓRICAS E REFERÊNCIAS

A Seção L.6 (disponível on-line) apresenta uma discussão sobre o Illiac IV (um representante das primeiras arquiteturas SIMD) e o Cray-1 (um representante das arquiteturas vetoriais). Nós também examinamos as extensões SIMD multimídia e a história das GPUs.

ESTUDO DE CASO E EXERCÍCIOS POR JASON D. BAKOS

Estudo de caso: implementando um kernel vetorial em um processador vetorial e GPU

Conceitos ilustrados neste estudo de caso

- Programação de processadores vetorial
- Programação de GPUs
- Estimativa de desempenho

MrBayes é uma popular e conhecida aplicação computacional para biologia para inferir os históricos evolucionários entre um conjunto de espécies de entrada com base nos seus dados de sequência de DNA multiplamente alinhados de comprimento n. O MrBayes funciona realizando uma busca heurística nos espaço de todas as topologias de árvore binária, na qual as entradas são as folhas. Para avaliar uma árvore em particular, a aplicação deve calcular uma tabela de probabilidade $n \times 4$ (chamada clP) para cada nó interno. A tabela é uma função das tabelas de probabilidade condicionais dos dois nós descendentes do nó (clL e clR, ponto flutuante de precisão simples) e suas tabelas de probabilidade de transição associadas $n \times 4 \times 4$ (tiPL e tiPR, ponto flutuante de precisão simples). Um dos kernels dessa aplicação é o cálculo dessa tabela de probabilidade condicional e é mostrado a seguir:

```
para (k=0; k<seq_length; k++) {
   clP[h++] = (tiPL[AA]*clL[A] + tiPL[AC]*clL[C] + tiPL[AG]*clL[G] + tiPL[AT]*clL[T])
   *(tiPR[AA]*clR[A] + tiPR[AC]*clR[C] + tiPR[AG]*clR[G] + tiPR[AT]*clR[T]);
   clP[h++] = (tiPL[CA]*clL[A] + tiPL[CC]*clL[C] + tiPL[CG]*clL[G] + tiPL[CT]*clL[T])
   *(tiPR[CA]*clR[A] + tiPR[CC]*clR[C] + tiPR[CG]*clR[G] + tiPR[CT]*clR[T]);
   clP[h++] = (tiPL[GA]*clL[A] + tiPL[GC]*clL[C] + tiPL[GG]*clL[G] + tiPL[GT]*clL[T])
   *(tiPR[GA]*clR[A] + tiPR[GC]*clR[C] + tiPR[GG]*clR[G] + tiPR[GT]*clR[T]);
   clP[h++] = (tiPL[TA]*clL[A] + tiPL[TC]*clL[C] + tiPL[TG]*clL[G] + tiPL[TT]*clL[T])
   *(tiPR[TA]*clR[A] + tiPR[TC]*clR[C] + tiPR[TG]*clR[G] + tiPR[TT]*clR[T]);
   clL += 4;
   clR += 4;
   tiPL += 16;
   tiPR += 16;
}
```

Constantes	Valores
AA,AC,AG,AT	0,1,2,3
CA,CC,CG,CT	4,5,6,7
GA,GC,GG,GT	8,9,10,11
TA,TC,TG,TT	12,13,14,15
A,C,G,T	0,1,2,3

FIGURA 4.32 Constantes e valores para o estudo de caso.

4.1 [25] <4.2, 4.3> Considere as constantes mostradas na Figura 4.32. Mostre o código para MIPS e VMIPS. Considere que não podemos usar carregamentos ou armazenamentos scatter-gather. Considere que os endereços iniciais de `tiPL`, `tiPR`, `clL`, `clP` e `CIP` estão em `RtiPL`, `RtiPR`, `RclR` e `RclP`, respectivamente. Considere que o comprimento do registrador VMIPS é programável pelo usuário e pode ser designado configurando o registrador especial VL (p. ex., li VL 4). Para facilitar as reduções de adição vetorial, considere que adicionamos as seguintes instruções ao VIMPS:

`SUMR.S Fd, Vs Redução de somatório vetorial precisão simples:`

Esta instrução realiza uma redução de somatório em um registrador vetorial `Vs`, gravando a soma no registrador escalar `Fd`.

4.2 [5] <4.2, 4.3> Considerando `seq_length` == 500, qual é a contagem de instruções dinâmicas para as duas implementações?

4.3 [25] <4.2, 4.3> Considere que a instrução de redução vetorial seja executada na unidade fundamental vetorial, similar a uma instrução soma vetorial. Mostre como a sequência de código estabelece comboios supondo uma única instância de cada unidade funcional vetorial. Quantos chimes o código vai requerer? Quantos ciclos por FLOP são necessários, ignorando o overhead de despacho de instrução vetorial?

4.4 [15] <4.2, 4.3> Agora considere que possamos usar carregamentos e armazenamentos scatter-gather (LVI e SVI). Considere que `tiPL`, `tiPR`, `clL`, `clR` e `clP` são posicionados consecutivamente na memória. Por exemplo, se `if seq_length` = =500, o array tiPR começaria 500*4 bytes depois do array tiPL. Como isso afeta o modo como você pode gravar o código VMIPS para esse kernel? Considere que você possa inicializar os registradores vetoriais usando a seguinte técnica que, por exemplo, inicializaria o registrador vetorial V1 com os valores (0,0;2000;2000):

```
LI R2,0
SW R2,vec
SW R2,vec+4
LI R2,2000
SW R2,vec+8
SW R2,vec+12
LV V1,vec
```

Considere que o comprimento máximo vetorial é 64. Há algum modo como o desempenho pode ser melhorado usando carregamentos gather-scatter? Se sim, quanto?

4.5 [25] <4.4> Agora considere que queremos implementar o kernel MrBayes em uma GPU usando um único bloco de threads. Reescreva o código *C* para o kernel usando CUDA. Considere que os ponteiros para as tabelas de probabilidade condicional

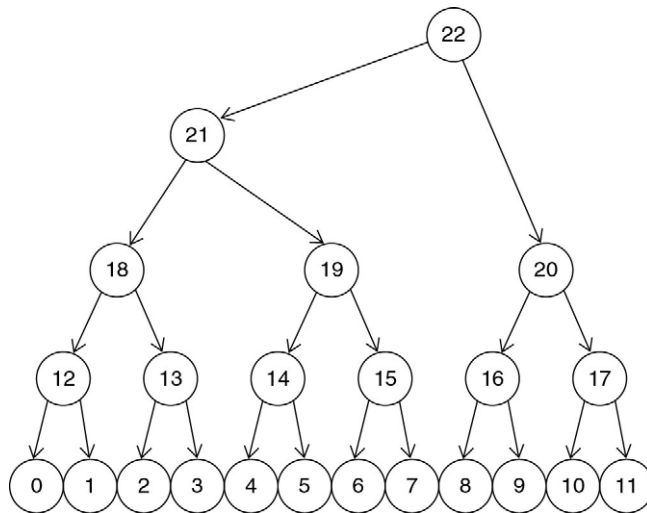



FIGURA 4.33 Árvore de amostras.

e probabilidade de transição são especificados como parâmetros para o kernel. Invoque um thread para cada iteração do loop. Carregue quaisquer valores reutilizados na memória compartilhada antes de realizar operações sobre eles.

4.6 [15] <4.4> Com CUDA podemos usar paralelismo de granularidade grossa no nível de bloco para calcular as probabilidades condicionais de múltiplos nós em paralelo. Considere que queremos calcular as probabilidades condicionais da base da árvore para cima. Considere que os arrays de probabilidade condicional e probabilidade de transição são organizados na memória como descrito na questão 4, e o grupo de tabelas para cada um dos 12 nós de folha é armazenado em posições de memória consecutivas na ordem do número de nó. Considere ainda que queremos calcular a probabilidade condicional para os nós 12 a 17, como mostrado na Figura 4.33. Mude o método como você calcula os índices de array em sua resposta para o Exercício 4.5 a fim de incluir o número de bloco.

4.7 [15] <4.4> Converta seu código do Exercício 4.6 em código PTX. Quantas instruções são necessárias para o kernel?

4.8 [10] <4.4> Quão bem você espera que esse código seja realizado em uma GPU? Explique sua resposta.

Exercícios

4.9 [10/20/20/15/15] <4.2> Considere o código a seguir, que multiplica dois vetores que contêm valores complexos de precisão simples:

```
para (i=0;i<300;i++) {
      c_re[i] = a_re[i] * b_re[i] - a_im[i] * b_im[i];
      c_im[i] = a_re[i] * b_im[i] + a_im[i] * b_re[i];
}
```

Considere que o processador roda a 700 MHz e tem um comprimento máximo de vetor de 64. A unidade de carregamento/armazenamento tem um overhead de inicialização de 15 ciclos, a unidade de multiplicação, oito ciclos, e a unidade de soma/subtração, cinco ciclos.

a. [10] <4.2> Qual é a intensidade aritmética desse kernel? Justifique sua resposta.

b. [20] <4.2> Converta esse loop em código assembly VMIPS usando strip mining.

c. [20] <4.2> Considere encadeamento e um único pipeline de memória. Quantos chimes são necessários? Quantos ciclos de clock são requeridos por valor de resultado complexo, incluindo overhead de inicialização?

d. [15] <4.2> Se a sequência vetorial for encadeada, quantos ciclos de clocks serão necessários por valor de resultado complexo, incluindo o overhead?

e. [15] <4.2> Agora suponha que o processador tenha três pipelines de memória e encadeamento. Se não houver conflitos de banco nos acessos de loop, quantos ciclos de clock serão necessários por resultado?

4.10 [30] <4.4> Neste problema, vamos comparar o desempenho de um processador vetorial com um sistema híbrido que contém um processador escalar e um coprocessador baseado em GPU. No sistema híbrido, o processador host tem um desempenho escalar superior ao da GPU, então nesse caso todo o código escalar é executado no processador host, enquanto todo o código vetorial é executado na GPU. Vamos nos referir ao primeiro sistema como computador vetorial e ao segundo sistema como computador híbrido. Considere que sua aplicação-alvo contém um kernel vetorial com intensidade aritmética de 0,5 FLOP por byte da DRAM acessado. Entretanto, a aplicação também tem um componente escalar que deve ser realizado antes e depois do kernel para preparar os vetores de entrada e os vetores de saída, respectivamente. Para um conjunto de dados de amostra, a porção escalar do código requer 400 ms de tempo de execução no processador vetorial e no processador host no sistema híbrido. O kernel lê vetores de entrada consistindo em 200 MB de dados e tem dados de saída consistindo em dados de 100 MB. O processador vetorial tem um pico de largura de banda de memória de 30 GB/s e a GPU tem um pico de largura de banda de memória de 150 GB/s. O sistema híbrido tem um overhead adicional que requer todos os vetores de entrada a serem transferidos entre a memória do host e a memória local da GPU antes e depois do kernel ser invocado. O sistema híbrido tem uma largura de banda de acesso direto à memória (DMA) de 10 GB/s e uma latência média de 10 ms. Considere que o processador vetorial e a GPU têm desempenho limitado pela largura de banda da memória. Calcule o tempo de execução requerido pelos dois computadores para esta aplicação.

4.11 [15/25/25] <4.4, 4.5> A Seção 4.5 discutiu a operação de redução que reduz um vetor para um escalar por aplicação repetida de uma operação. Uma redução é um tipo especial de recorrência de loop. Mostramos a seguir um exemplo:

```
dot=0.0;
para (i=0;i<64;i++) dot = dot + a[i] * b[i];
```

Um compilador de vetorização pode aplicar uma transformação chamada *expansão escalar*, que expande um escalar em um vetor e divide o loop de modo que a multiplicação possa ser realizada com uma operação vetorial, deixando a redução como uma operação escalar separada:

```
para (i=0;i<64;i++) dot[i] = a[i] * b[i];
para (i=1;i<64;i++) dot[0] = dot[0] + dot[i];
```

Como mencionado na Seção 4.5, se permitirmos que a soma de ponto flutuante seja associativa, existem diversas técnicas disponíveis para paralelizar a redução.

a. [15] <4.4, 4.5> Uma técnica é chamada a dobrar a recorrência, que soma sequências de vetores progressivamente mais curtos (p. ex., dois vetores de 32 elementos, então dois vetores de 16 elementos, e assim por diante). Mostre como seria o código C para executar o segundo loop desse modo.

b. [25] <4.4, 4.5> Em alguns processadores vetoriais, os elementos individuais dentro dos registradores vetoriais são endereçáveis; nesse caso, os operandos

de uma operação vetorial podem ter duas partes diferentes do mesmo registrador vetorial. Isso permite outra solução para a redução chamada *somas parciais.* A ideia é reduzir o vetor a *m* somas, em que *m* é a latência total na unidade funcional vetorial, incluindo os tempos de leitura e gravação de operandos. Considere que os registradores vetoriais VMIPS sejam endereçáveis (p. ex., você pode iniciar uma operação vetorial com o operando V1(16), indicando que o operando de entrada começa com o elemento 16). Considere também que a latência total para somas, incluindo a leitura e operandos e a gravação do resultado, é de oito ciclos. Escreva uma sequência de código VMIPS que reduza o conteúdo de V1 para oito somas parciais.

c. [25] <4.4, 4.5> Ao realizar uma redução em uma GPU, um thread é associado a cada elemento no vetor de entrada. O primeiro passo é cada thread gravar seu valor correspondente na memória compartilhada. A seguir, cada thread entra em um loop que soma cada par de valores de entrada. Isso reduz o número de elementos à metade após cada iteração, significando que o número de threads ativos também é reduzido à metade a cada iteração. Para maximizar o desempenho da redução, o número de warps totalmente preenchidos deve ser maximizado durante o curso do loop. Em outras palavras, os threads ativos devem ser contíguos. Além disso, cada thread deve indexar o array compartilhado de tal modo que evite conflitos de banco com a memória compartilhada. O loop a seguir viola somente a primeira dessas instruções e também usa o operador módulo, muito dispendioso para as GPUS:

```
unsigned int tid = threadIdx.x;
para(unsigned int s=1; s < blockDim.x; s *= 2) {
if ((tid % (2*s)) = = 0) {
sdata[tid] += sdata[tid + s];
}
__syncthreads();
}
```

Reescreva o loop para atender a essas regras e elimine o uso do operador módulo. Considere que existam 32 threads por warp e um conflito de banco ocorre sempre que dois ou mais threads do mesmo warp referenciam um índice cujos módulos por 32 sejam iguais.

4.12 [10/10/10/10] <4.3> O kernel a seguir realiza uma porção do método *finite-difference time-domain* (FDTD) para calcular as equações de Maxwell em um espaço tridimensional para um dos benchmarks SPEC06fp:

```
para (int x=0; x<NX-1; x++) {
para (int y=0; y<NY-1; y++) {
para (int z=0; z<NZ-1; z++) {
int index = x*NY*NZ + y*NZ + z;
if (y>0 && x >0) {
material = IDx[index];
dH1 = (Hz[index] - Hz[index-incrementY])/dy[y];
dH2 = (Hy[index] - Hy[index-incrementZ])/dz[z];
Ex[index] = Ca[material]*Ex[index]+Cb[material]*(dH2-dH1);
}}}}
```

Considere que dH1, dH2, Hy, Hz, dy, dz, Ca, Cb e Ex sejam arrays de ponto flutuante de precisão simples. Considere também que IDx é um array de inteiros sem sinal.

a. [10] <4.3> Qual é a intensidade aritmética desse kernel?

b. [10] <4.3> Esse kernel é adequado para execução vetorial ou SIMD? Por quê?

c. [10] <4.3> Considere que esse kernel será executado em um processador que tem largura de banda de memória de 30 GB/s. Esse kernel será limitado pela memória ou computacionalmente?

d. [10] <4.3> Desenvolva um modelo roofline para esse processador considerando que ele tenha um pico de throughput computacional de 85 GFLOP/s.

4.13 [10/15] <4.4> Considere uma arquitetura de GPU que contenha 10 processadores SIMD. Cada instrução SIMD tem uma largura de 32 e cada processador SIMD contém oito (pistas para aritmética de precisão simples e instruções de carregamento/armazenamento, significando que cada instrução SIMD não divergida pode produzir 32 resultados a cada quatro ciclos). Considere um kernel que tenha desvios divergentes que faça com que uma média de 80% dos threads estejam ativos. Considere que 70% de todas as instruções SIMD executadas sejam aritméticas de precisão simples e 20% sejam carregamento/armazenamento. Uma vez que nem todas as latências de memória são cobertas, considere uma taxa média de despacho de instruções SIMD de 0,85. Considere que a GPU tem uma velocidade de clock de 1,5 GHz.

a. [10] <4.4> Calcule o throughput, em GFLOP/s, para esse kernel nessa GPU.

b. [15] <4.4> Considere que você tem as seguintes opções:

- Aumentar o número de pistas de precisão simples para 16.
- Aumentar o número de processadores SIMD para 15 (considere que essa mudança não afeta outras medidas de desempenho e que o código está em proporção com os processadores adicionais).
- Adicionar um cache que vai efetivamente reduzir a latência de memória em 40%, o que vai aumentar a taxa de despacho de instruções para 0,95.

Qual é o ganho de velocidade em throughput para cada uma dessas melhorias?

4.14 [10/15/15] <4.5> Neste exercício, vamos examinar diversos loops e analisar seu potencial para paralelização.

a. [10] <4.5> O loop a seguir possui uma dependência carregada pelo loop?

```
para (i=0;i<100;i++) {
 A[i] = B[2*i+4];
 B[4*i+5] = A[i];
 }
```

b. [15] <4.5> No loop a seguir, encontre todas as dependências reais, dependências de saída e antidependências. Elimine as dependências de saída e antidependências por renomeação.

```
para (i=0;i<100;i++) {
 A[i] = A[i] * B[i]; /* S1 */
 B[i] = A[i] + c; /* S2 */
 A[i] = C[i] * c; /* S3 */
 C[i] = D[i] * A[i]; /* S4 */
```

c. [15] <4.5> Considere o seguinte loop:

```
para (i=0;i < 100;i++) {
 A[i] = A[i] + B[i]; /* S1 */
 B[i+1] = C[i] + D[i]; /* S2 */
}
```

Existem dependências entre S1 e S2? Esse loop é paralelo? Se não, mostre como torná-lo paralelo.

4.15 [10] <4.4> Liste e descreva pelo menos quatro fatores que influenciam o desempenho dos kernels de GPU. Em outras palavras, que comportamento de tempo de execução causado pelo código do kernel provoca uma redução na utilização de recursos durante a execução do kernel?

4.16 [10] <4.4> Considere uma GPU hipotética com as seguintes características:

- Taxa de clock de 1,5 GHz
- Contém 16 processadores SIMD, cada qual contendo 16 unidades de ponto flutuante de precisão simples
- Possui 100 GB/s de largura de banda de memória fora do chip

Sem considerar a largura de banda de memória, qual é o pico de throughput de ponto flutuante de precisão simples para essa GPU em GFLOP/s, considerando que todas as latências de memória podem ser ocultadas? Esse throughput é sustentável, dada a limitação em largura de banda de memória?

4.17 [60] <4.4> Para este exercício de programação, você vai escrever e caracterizar o comportamento de um kernel CUDA que contenha muito paralelismo em nível de dados, mas também comportamento de execução condicional. Use o toolkit NVIDIA CUDA juntamente com o GPU-SIM da Universidade da Colúmbia Britânica (*www.ece.ubc.ca/~aamodt/gpgpu-sim/*) ou o CUDA Profiler para escrever e compilar um kernel CUDA que realize 100 iterações do *Conway's Game of Life* para um tabuleiro de 256 × 256 e retorne ao status final do tabuleiro de jogo para o host. Considere que o tabuleiro é inicializado pelo host. Associe um thread a cada célula. Tenha a certeza de adicionar uma barreira após cada iteração do jogo. Use as seguintes regras de jogo:

- Qualquer célula viva com menos de dois vizinhos vivos morre.
- Qualquer célula viva com dois ou três vizinhos vivos vai para a próxima geração.
- Qualquer célula viva com mais de três vizinhos vivos morre.
- Qualquer célula morta com exatamente três vizinhos vivos se torna uma célula viva.

Depois de acabar o kernel, responda às seguintes perguntas:

a. [60] <4.4> Compile seu código usando a opção –ptx e inspecione a representação PTC do seu kernel. Quantas instruções PTX compõem a implementação PTX do seu kernel? As seções condicionais do seu kernel incluem instruções de desvio ou somente instruções sem desvio com predicado?

b. [60] <4.4> Depois de executar seu código no simulador, qual é a contagem dinâmica de instruções? Quais são as *instruções por ciclo* (IPC) alcançado ou taxa de despacho de instruções? Qual é o detalhamento dinâmico de instruções em termos de instruções de controle, instruções da unidade lógico-aritmética (ALU) e instruções de memória? Existem conflitos de banco de memória compartilhados? Qual é a largura de banda de memória efetiva fora do chip?

c. [60] <4.4> Implemente uma versão melhorada do seu kernel em que as referências à memória fora do chip sejam reunidas e observe as diferenças em desempenho de tempo de execução.

Paralelismo em nível de thread

A virada da organização convencional veio em meados da década de 1960, quando a lei dos rendimentos decrescentes começou a ter efeito sobre o esforço de aumentar a velocidade operacional de um computador [...] Os circuitos eletrônicos são, por fim, limitados em sua velocidade de operação pela velocidade da luz [...] e muitos dos circuitos já estavam operando na faixa do nanossegundo.

W. Jack Bouknight *et al.*, The Illiac IV System (1972)

Dedicamos todo o futuro desenvolvimento de produtos aos projetos multicore. Acreditamos que esse seja um ponto-chave de inflexão para o setor.

Paul Otellini, *presidente da Intel,* *ao descrever os futuros rumos da empresa no Intel Developers Forum, em 2005*

5.1 INTRODUÇÃO

Conforme mostram as citações que abrem este capítulo, a visão de que os avanços na arquitetura de uniprocessador chegavam a um fim tem sido mantida por alguns pesquisadores há muitos anos. Obviamente, essas visões foram prematuras; na verdade, durante o período de 1986 a 2003, o crescimento do desempenho do uniprocessador cresceu, conduzido pelos microprocessadores, atingindo a taxa mais alta desde os primeiros computadores transistorizados, no final da década de 1950 e início da década de 1960.

Apesar disso, a importância dos multiprocessadores aumentou por toda a década de 1990, enquanto os projetistas buscavam um meio de criar servidores e supercomputadores que alcançassem desempenho mais alto do que um único microprocessador, enquanto exploravam as tremendas vantagens no custo-desempenho dos microprocessadores como

commodity. Conforme discutimos nos Capítulos 1 e 3, o atraso no desempenho do uniprocessador que surgiu dos rendimentos decrescentes na exploração do ILP, combinado com a crescente preocupação com a potência, está levando a uma nova era na arquitetura de computador — uma era na qual os multiprocessadores desempenham um papel importante. A segunda citação captura esse evidente ponto de inflexão.

Essa tendência em direção a mais confiança no multiprocessamento é reforçada por outros fatores:

- As eficiências drasticamente menores no uso de silício e energia que foram encontradas entre 2000 e 2005, quando os projetistas tentaram encontrar e explorar mais ILP, o que se mostrou ineficiente, já que os custos da energia e do silício cresceram mais do que o desempenho. Além do ILP, o único modo escalável e de uso geral que conhecemos para aumentar o desempenho mais rápido do que a tecnologia básica permite (de uma perspectiva de chaveamento) é o multiprocessamento.
- Um crescente interesse por servidores de alto nível, conforme a computação em nuvem e o "software sob demanda" se tornam mais importantes.
- Um crescimento nas aplicações orientadas a computação intensiva de dados, pela disponibilidade de grande quantidade de dados na internet.
- A percepção de que o desempenho crescente no desktop é menos importante (fora dos gráficos, pelo menos), seja porque o desempenho atual é aceitável, seja porque aplicações altamente pesadas computacionalmente ou em termos de dados estão sendo feitas na nuvem.
- Compreensão melhorada de como usar multiprocessadores de modo eficiente, especialmente nos ambientes de servidor em que existe significativo paralelismo natural, vindo de grandes conjuntos de dados, paralelismo natural (que ocorre em códigos científicos) ou paralelismo entre grande número de requisições independentes (paralelismo em nível de requisição).
- As vantagens de aproveitar um investimento de projeto pela replicação, em vez de um projeto exclusivo — todos os projetos de multiprocessador oferecem tal aproveitamento.

Neste capítulo, nos concentraremos em explorar o paralelismo em nível de thread (TLP). O TLP implica a existência de múltiplos contadores de programa e, portanto, é explorado primeiramente através de MIMDs. Embora as MIMDs estejam por aí há décadas, o movimento do paralelismo de nível de thread para o primeiro plano de toda a computação, de aplicações embarcadas a servidores de alto desempenho, é relativamente recente. Do mesmo modo, o uso extensivo de paralelismo em nível de thread para aplicações de uso geral no lugar de aplicações científicas é relativamente novo.

Nosso foco, neste capítulo, recairá sobre os *multiprocessadores*, que definimos como computadores consistindo em processadores fortemente acoplados cuja coordenação e cujo uso costumam ser controlados por um único sistema operacional que compartilha memória através de um espaço de endereços compartilhado. Tais sistemas exploram o paralelismo em nível de thread através de dois modelos de software diferentes. O primeiro é a execução de um conjunto de threads fortemente acoplados colaborando em uma única tarefa, em geral chamado *processamento paralelo.* O segundo é a execução de múltiplos processos relativamente independentes que podem se originar de um ou mais usuários, o que é uma forma de *paralelismo em nível de requisição*, embora em uma escala muito menor do que a que exploraremos no Capítulo 6. O paralelismo em nível de requisição pode ser explorado por uma única aplicação sendo executada em múltiplos processadores, como uma base de dados respondendo a pesquisas, ou múltiplas aplicações rodando independentemente, muitas vezes chamado *multiprogramação.*

Os multiprocessadores que vamos examinar neste capítulo costumam variar em tamanho, indo desde dois processadores até dúzias de processadores, e que se comunicam e se coordenam através do compartilhamento de memória. Embora compartilhar através da memória implique um espaço de endereços compartilhado, não significa necessariamente

que exista uma única memória física. Tais multiprocessadores incluem sistemas de chip único com múltiplos núcleos, chamados *multicore*[1] e computadores consistindo em múltiplos chips, cada qual podendo ter um projeto multicore.

Além dos multiprocessadores reais, vamos retornar ao tópico do multithreading, uma técnica que suporta múltiplos threads sendo executados de modo interligado em um único processador de despacho múltiplo. Muitos processadores multicore também incluem suporte para multithreading.

No Capítulo 6, consideraremos computadores de ultraescala construídos a partir de um grande número de processadores. Esses sistemas de larga escala costumam ser usados para computação em nuvem com um modelo que supõe grande número de requisições independentes ou tarefas computacionais paralelas e intensas. Quando esses clusters aumentam para dezenas de milhares de servidores, nós os chamamos *computadores em escala warehouse.*

Além dos multiprocessadores que estudaremos aqui e os sistemas em escala warehouse do Capítulo 6, existe uma gama especial de sistemas multiprocessadores de grande escala, às vezes chamados *multicomputadores.* Eles não são tão fortemente acoplados quanto os multiprocessadores examinados neste capítulo, mas são mais fortemente acoplados do que os sistemas em escala warehouse do próximo. Tais multicomputadores são usados principalmente em cálculos científicos de alto nível. Muitos outros livros, como o de Culler, Singh e Gupta (1999), abordam esses sistemas em detalhes. Devido à natureza grande e mutante do campo do multiprocessamento (o livro citado tem mais de 1.000 páginas e trata *apenas de microprocessadores*!), nós decidimos concentrar nossa atenção no que consideramos as partes mais importantes e de uso mais geral do espaço de computação. O Apêndice I aborda alguns dos problemas que surgem na construção desses computadores no contexto de aplicações científicas de alta escala.

Assim, nosso foco recairá sobre os multiprocessadores com número pequeno a médio de processadores (2 a 32). Esses projetos dominam em termos de unidades e valores monetários. Só daremos um pouco de atenção de projeto de multiprocessador em escala maior (33 ou mais processadores) principalmente no Apêndice I, que abrange mais aspectos do projeto desses processadores, além do desempenho de comportamento para cargas de trabalho científicas paralelas, uma classe primária de aplicações para multiprocessadores em grande escala. Nos multiprocessadores em grande escala, as redes de interconexão são uma parte crítica do projeto; o Apêndice F aborda esse tópico.

Arquitetura de multiprocessadores: problemas e abordagem

Para tirar proveito de um multiprocessador MIMD com *n* processadores, precisamos ter pelo menos *n* threads ou processos para executar. Os threads independentes dentro de um único processo geralmente são identificados pelo programador ou criados pelo compilador. Os threads podem vir de processos em grande escala, independentes, escalonados e manipulados pelo sistema operacional. No outro extremo, um thread pode consistir em algumas dezenas de iterações de um loop, geradas por um compilador paralelo que explora o paralelismo de dados no loop. Embora a quantidade de computação atribuída a um thread, chamada *tamanho de granularidade*, seja importante na consideração de como explorar de forma eficiente o paralelismo em nível de thread, a distinção qualitativa importante entre o paralelismo e o nível de instrução é que o paralelismo em nível de thread é identificado em alto nível pelo sistema de software e os threads consistem em centenas a milhões de instruções que podem ser executadas em paralelo.

Os threads também podem ser usados para explorar o paralelismo em nível de dados, embora o overhead provavelmente seja mais alto do que seria visto em um computador

[1] **Nota da Tradução:** A tradução do termo *multicore* seria "multinúcleos". Porém, como esse tipo de processador ficou conhecido comercialmente como *multicore*, resolvemos manter o termo em inglês.

SIMD ou em uma GPU. Esse overhead significa que a granularidade precisa ser suficientemente grande para explorar o paralelismo de modo eficiente. Por exemplo, embora um processador vetorial (Cap. 4) possa ser capaz de colocar operações em paralelo de forma eficiente em vetores curtos, a granularidade resultante quando o paralelismo é dividido entre muitos threads pode ser tão pequena a ponto de o overhead tornar a exploração do paralelismo proibitivamente dispendiosa em um MIMD.

Os multiprocessadores MIMD existentes estão em duas classes, dependendo do número de processadores envolvidos, que, por sua vez, dita uma organização de memória e uma estratégia de interconexão. Vamos nos referir aos multiprocessadores por sua organização de memória, pois o que constitui um número pequeno ou grande de processadores provavelmente mudará com o tempo.

O primeiro grupo, que chamamos *multiprocessadores simétricos* (*memória compartilhada*) (*SMPs*) *ou multiprocessadores centralizados de memória compartilhada*, tem um número pequeno de núcleos, geralmente oito ou menos. Para multiprocessadores com pouca quantidade de processadores, é possível que os processadores compartilhem uma única memória centralizada à qual todos os processadores tenham igual acesso; daí o termo *simétrico*. Em chips multicore, a memória é efetivamente compartilhada de modo centralizado entre os núcleos, e todos os multicores existentes são SMPs. Quando mais de um multicore é conectado, existem memórias separadas para cada multicore, então a memória é distribuída em vez de centralizada.

Às vezes, as arquiteturas SMP também são chamadas *multiprocessadores de acesso uniforme à memória* (UMA), advindo do fato de que todos os processadores possuem uma latência de memória uniforme, mesmo que essa memória seja organizada em múltiplos bancos. A Figura 5.1 mostra como são esses multiprocessadores. A arquitetura dos SMPs é o assunto da Seção 5.2, em que explicaremos a técnica no contexto de um multicore.

A técnica de projeto alternativa consiste em multiprocessadores com memória fisicamente distribuída, chamada *memória compartilhada distribuída* (distributed shared memory — DSM). A Figura 5.2 mostra como se parecem esses multiprocessadores. Para dar suporte a uma grande quantidade de processadores, a memória precisa ser distribuída entre os processadores em vez de centralizada; caso contrário, o sistema de memória não será capaz de dar suporte às demandas de largura de banda de um número maior de processadores sem incorrer em uma latência de acesso excessivamente longa. Com o rápido aumento no desempenho do processador e associado ao aumento nos requisitos de largura de banda da memória de um processador, o tamanho de um multiprocessador para o qual a memória distribuída é preferida continua a diminuir. O grande número de processadores também aumenta a necessidade de uma interconexão com alta largura de banda, da qual veremos exemplos no Apêndice F. Tanto as redes diretas (ou seja, switches) quanto as redes indiretas (normalmente malhas multidimensionais) são usadas.

A distribuição da memória entre os nós aumenta a largura de banda e reduz a latência da memória. Um multiprocessador DSM é também chamado *NUMA* (acesso não uniforme à memória), já que o tempo de acesso depende da localização de uma palavra de dados na memória. As desvantagens principais para um DSM são que comunicar dados entre processadores se torna um pouco mais complexo e que um DSM requer mais esforço no software para tirar vantagem da largura de banda de memória maior gerada pelas memórias distribuídas. Já que todos os multiprocessadores multicore com mais de um chip processador (ou soquete) usam memória distribuída, vamos explicar a operação dos multiprocessadores de memória distribuída a partir desse ponto de vista.

Nas arquiteturas SMP e DSM, a comunicação entre threads ocorre através de um espaço de endereços compartilhado, o que significa que uma referência de memória pode ser feita por qualquer processador para qualquer local na memória, supondo que ele tenha

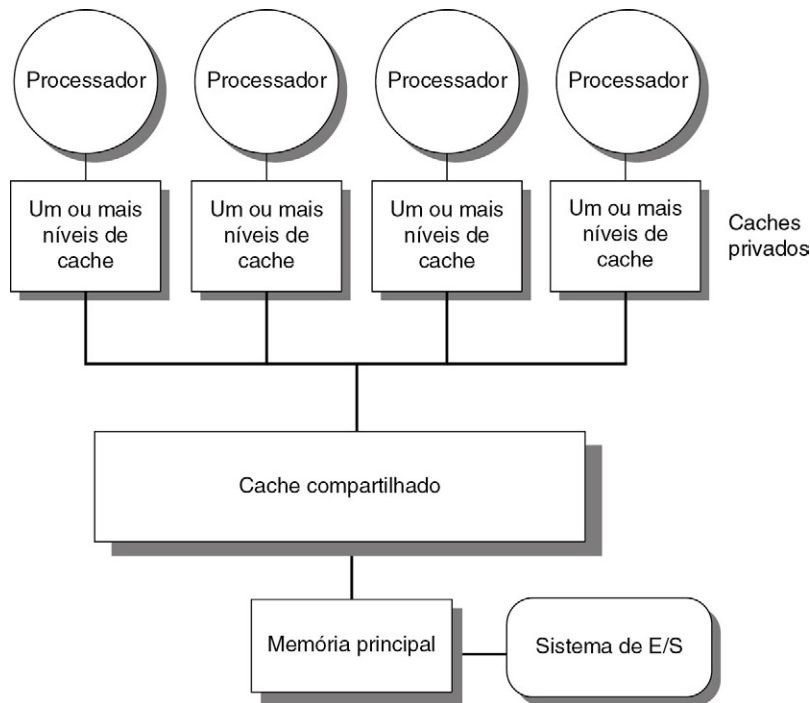



FIGURA 5.1 **Estrutura básica de um multiprocessador de memória compartilhada centralizada.**
Subsistemas de cache de múltiplos processadores compartilham a mesma memória física, normalmente conectada por um ou mais barramentos ou um switch. A principal propriedade da arquitetura é o tempo de acesso uniforme a toda a memória a partir de todos os processadores. Em uma versão multichip de cache compartilhada seria omitida, e o barramento ou rede de interconexão conectando os processadores à memória seria executado entre chips, em vez de dentro de um único chip.

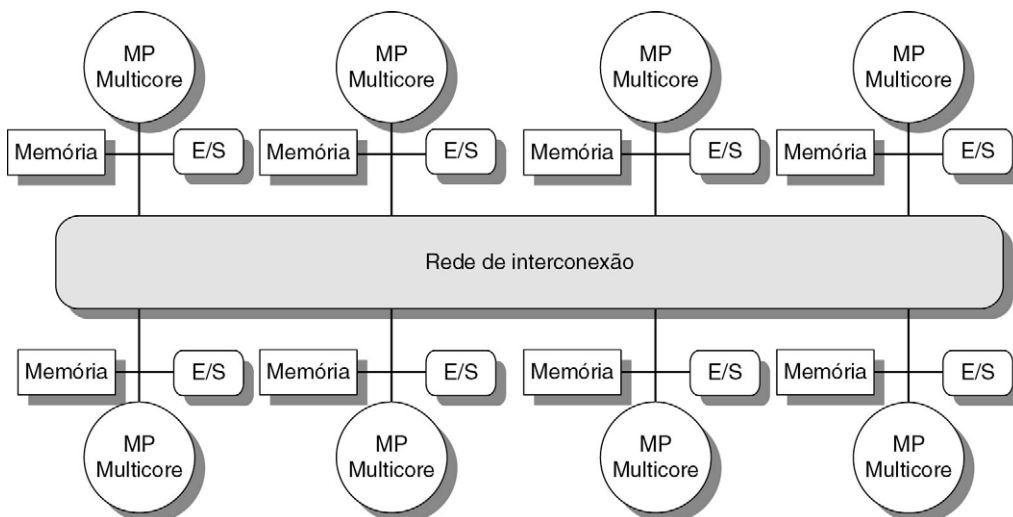



FIGURA 5.2 **A arquitetura básica de um multiprocessador de memória distribuída em 2011 consiste em um chip de multiprocessador multicore com memória e possivelmente E/S anexas e uma interface para uma rede de interconexão que conecta todos os nós.**
Cada núcleo de processador compartilha toda a memória, embora o tempo de acesso para a memória anexada ao chip do núcleo seja muito mais rápido do que o tempo de acesso às memórias remotas.

os direitos de acesso corretos. O nome *memória compartilhada*, associado tanto ao SMP quanto ao DSM, se refere ao fato de que o *espaço de endereços* é compartilhado.

Em contraste, os clusters e computadores em escala warehouse do Capítulo 6 se parecem com computadores individuais conectados por uma rede, e a memória de um processador não pode ser acessada por outro processador sem a assistência de protocolos de software sendo executados nos dois processadores. Em tais projetos, protocolos de envio de mensagem são usados para comunicar dados entre os processadores.

Desafios do processamento paralelo

A aplicação dos multiprocessadores varia desde a execução de tarefas independentes, essencialmente sem comunicação, até a execução de programas paralelos em que os threads precisam se comunicar para completar a tarefa. Dois obstáculos importantes, ambos explicáveis pela lei de Amdahl, tornam o processamento paralelo desafiador. O grau pelo qual esses obstáculos são difíceis ou fáceis é determinado tanto pela aplicação quanto pela arquitetura.

O primeiro obstáculo tem a ver com o paralelismo limitado disponível nos programas, e o segundo surge do custo relativamente alto das comunicações. As limitações no paralelismo disponível tornam difícil alcançar bons ganhos de velocidade em qualquer processador paralelo, como mostra nosso primeiro exemplo.

Exemplo Suponha que você queira alcançar um ganho de velocidade de 80 com 100 processadores. Que fração da computação original pode ser sequencial?

Resposta Conforme vimos no Capítulo 1, a lei de Amdahl é

$$\text{Ganho de velocidade} = \frac{1}{\dfrac{\text{Fração}_{\text{melhorado}}}{\text{Ganho de velocidade}_{\text{melhorado}}} + (1 - \text{Fração}_{\text{melhorado}})}$$

Para simplificar, considere que o programa opere em apenas dois modos: paralelo com todos os processadores totalmente usados, que é o modo avançado, e serial, com apenas um processador em uso. Com essa simplificação, o ganho de velocidade no modo avançado é simplesmente o número de processadores, enquanto a fração do modo avançado é o tempo gasto no modo paralelo. Substituindo na equação anterior:

$$80 = \frac{1}{\dfrac{\text{Fração}_{\text{paralelo}}}{100} + (1 - \text{Fração}_{\text{paralelo}})}$$

Simplificando essa equação, temos:

$$0{,}8 \times \text{Fração}_{\text{paralelo}} + 80 \times (1 - \text{Fração}_{\text{paralelo}}) = 1$$

$$80 - 79{,}2 \times \text{Fração}_{\text{paralelo}} = 1$$

$$\text{Fração}_{\text{paralelo}} = \frac{80 - 1}{79{,}2}$$

$$\text{Fração}_{\text{paralelo}} = 0{,}9975$$

Assim, para alcançar um ganho de velocidade de 80 com 100 processadores, apenas 0,25% da computação original pode ser sequencial. Naturalmente, para conseguir ganho de velocidade linear (ganho de velocidade de n com n processadores), o programa inteiro precisa ser paralelo, sem partes seriais. Na prática, os programas não só operam no modo totalmente paralelo ou sequencial, como normalmente utilizam menos do que o complemento total de processadores ao executar no modo paralelo.

O segundo desafio importante no processamento paralelo envolve a grande latência do acesso remoto em um processador paralelo. Nos multiprocessadores de memória compartilhada existentes, a comunicação de dados entre os processadores pode custar 35-50 ciclos de clock (para múltiplos núcleos) até mais de 1.000 ciclos de clock (para multiprocessadores em grande escala), dependendo do mecanismo de comunicação, do tipo de rede de interconexão e da escala do multiprocessador. O efeito de longos atrasos de comunicação é claramente substancial. Vamos considerar um exemplo simples.

Exemplo Suponha uma aplicação executando em um multiprocessador com 32 processadores, que possui um tempo de 200 ns para lidar com a referência a uma memória remota. Para essa aplicação, considere que todas as referências, exceto aquelas referentes à comunicação, atingem a hierarquia de memória local, que é ligeiramente otimista. Os processadores ficam em stall em uma solicitação remota, e a frequência do processador é de 3,3 GHz. Se o CPI de base (considerando que todas as referências atingem a cache) é 0,5, quão mais rápido será o multiprocessador se não houver comunicação *versus* se 0,2% das instruções envolverem uma referência de comunicação remota?

Resposta É mais simples calcular primeiro o CPI. O CPI efetivo para o multiprocessador com 0,2% de referências remotas é

$$\text{CPI} = \text{CPI base} + \text{taxa de solicitação remota} \times \text{custo de solicitação remota}$$

$$= 0{,}5 + 0{,}2\% \times \text{custo de solicitação remota}$$

O custo de solicitação remota é

$$\frac{\text{Custo de acesso remoto}}{\text{Tempo de ciclo}} = \frac{200\,\text{ns}}{0{,}3\,\text{ns}} = 666\,\text{ciclos}$$

Logo, podemos calcular o CPI:

$$\text{CPI} = 0{,}5 + 1{,}2 = 1{,}7$$

O multiprocessador com todas as referências locais é 1,7/0,5 = 3,4 vezes mais rápido. Na prática, a análise de desempenho é muito mais complexa, pois alguma fração das referências que não são de comunicação se perderá na hierarquia local e o tempo de acesso remoto não terá um único valor constante. Por exemplo, o custo de uma referência remota poderia ser muito pior, pois a disputa causada por muitas referências que tentam usar a interconexão global poderia ocasionar atrasos maiores.

Esses problemas — paralelismo insuficiente e comunicação remota com longa latência — são os dois maiores desafios para o desempenho no uso dos microprocessadores. O problema do paralelismo de aplicação inadequado precisa ser atacado sobretudo no software com novos algoritmos que podem ter melhor desempenho paralelo, além de sistemas de software que maximizem o tempo gasto na execução com todos os processadores. A redução do impacto da longa latência remota pode ser atacada pela arquitetura e pelo programador. Por exemplo, podemos reduzir a frequência dos acessos remotos com mecanismos de hardware, como o caching de dados compartilhados ou com mecanismos de software, como a reestruturação dos dados para termos mais acessos locais. Podemos tentar tolerar a latência usando o multithreading (tratado mais adiante neste capítulo) ou a pré-busca (um tópico que abordamos extensivamente no Capítulo 2).

Grande parte deste capítulo enfoca as técnicas para reduzir o impacto da longa latência de comunicação remota. Por exemplo, as Seções 5.2 a 5.4 discutirão como o caching pode ser usado para reduzir a frequência de acesso remoto enquanto mantém uma visão coerente da memória. A Seção 5.5 discute o sincronismo, que, por envolver inerentemente a comunicação entre processadores e também limitar o paralelismo, é um importante gargalo em potencial. A Seção 5.6 abrange as técnicas de ocultação de latência e modelos consistentes de memória para a memória compartilhada. No Apêndice I, focalizamos principalmente os multiprocessadores em grande escala, que são usados predominantemente para o trabalho científico. Nesse apêndice, examinaremos a natureza de tais aplicações e os desafios de alcançar um ganho de velocidade com dezenas a centenas de processadores.

5.2 ESTRUTURAS DA MEMÓRIA COMPARTILHADA CENTRALIZADA

A observação de que o uso de grandes caches com múltiplos níveis pode reduzir substancialmente as demandas de largura de banda de memória de um processador é a principal percepção que motiva os multiprocessadores de memória centralizada. Originalmente, esses processadores eram todos de núcleo único e muitas vezes ocupavam uma placa inteira, e a memória estava localizada em um barramento compartilhado. Com processadores de alto desempenho mais recentes, as demandas de memória superaram a capacidade de barramentos razoáveis, e os microprocessadores recentes se conectam diretamente à memória, dentro de um único chip, o que às vezes é chamado *barramento backside ou de memória*, para distingui-lo do barramento usado para a conexão com as E/S. Acessar a memória local de um chip, seja para uma operação de E/S, seja para um acesso de outro chip, requer passar pelo chip "proprietário" dessa memória. Assim, o acesso à memória é assimétrico: mais rápido para a memória local e mais lento para a memória remota. Em um multicore, essa memória é compartilhada por todos os núcleos em um único chip, mas o acesso assimétrico à memória de um multicore a partir da memória de outro permanece.

As máquinas simétricas de memória compartilhada normalmente admitem o caching de dados compartilhados e privados. Os *dados privados* são usados por um único processador, enquanto os *dados compartilhados* são usados por múltiplos processadores, basicamente oferecendo comunicação entre os processadores através de leitura e escrita dos dados compartilhados. Quando um item privado é colocado na cache, seu local é migrado para a cache, reduzindo o tempo de acesso médio e também a largura de banda de memória exigida. Como nenhum outro processador usa os dados, o comportamento do programa é idêntico ao de um uniprocessador. Quando os dados compartilhados são colocados na cache, o valor compartilhado pode ser replicado em múltiplas caches. Além da redução na latência de acesso e largura de banda de memória exigida, essa replicação oferece uma redução na disputa que pode existir pelos itens de dados compartilhados que estão sendo lidos por múltiplos processadores simultaneamente. Contudo, o caching dos dados compartilhados introduz um novo problema: coerência da cache.

O que é coerência de cache de multiprocessador?

Infelizmente, o caching de dados compartilhados introduz um novo problema, pois a visão da memória mantida por dois processadores diferentes se dá através de seus caches individuais, que, sem quaisquer precauções adicionais, poderiam acabar vendo dois valores diferentes. A Figura 5.3 ilustra o problema e mostra como dois processadores diferentes podem ter dois valores diferentes para o mesmo local. Essa dificuldade geralmente é conhecida como *problema de coerência de cache*. Observe que o problema da coerência existe porque temos um estado global, definido primeiramente pela memória principal, e um estado local, definido pelas caches individuais, que são privativos para cada núcleo de processador. Assim, em um multicore em que algum nível de cache pode ser compartilhado (p. ex., um L3), embora alguns níveis sejam privados (p. ex., L1 e L2), o problema da coerência ainda existe e deve ser solucionado.

Informalmente, poderíamos afirmar que um sistema de memória será coerente se qualquer leitura de um item de dados retornar o valor escrito mais recentemente desse item de dados. Essa definição, embora intuitivamente atraente, é vaga e simplista; a realidade é muito mais complexa. Essa definição simples contém dois aspectos diferentes do comportamento do sistema de memória, ambos essenciais para a escrita de programas corretos de memória compartilhada. O primeiro aspecto, chamado *coerência*, define quais valores podem ser retornados por uma leitura. O segundo aspecto, chamado *consistência*, determina quando um valor escrito será retornado por uma leitura. Vejamos primeiro a coerência.

Tempo	Evento	Conteúdo da cache para o processador *A*	Conteúdo da cache para o processador *B*	Conteúdo da memória para a localização *X*
0				1
1	Processador *A* lê *X*	1		1
2	Processador *B* lê *X*	1	1	1
3	Processador *A* armazena 0 em *X*	0	1	0

FIGURA 5.3 O problema de coerência de cache para um único local de memória (*X*), lido e escrito por dois processadores (*A* e *B*).
Inicialmente, consideramos que nenhuma cache contém a variável e que *X* tem o valor 1. Também consideramos uma cache write-through; uma cache write-back acrescenta algumas complicações adicionais, porém semelhantes. Depois que o valor de *X* tiver sido escrito por *A*, a cache de *A* e a memória contêm o novo valor, mas não a cache de *B* e, se *B* ler o valor de *X*, ele receberá 1!

Um sistema de memória é coerente se

1. Uma leitura por um processador *P* a um local *X*, que acompanha uma escrita por *P* a *X*, sem escrita de *X* por outro processador ocorrendo entre a escrita e a leitura de *P*, sempre retorna o valor escrito por *P*.
2. Uma leitura por um processador ao local *X*, que acompanha uma escrita por outro processador a *X*, retornará o valor escrito se a leitura e a escrita forem suficientemente separadas no tempo e nenhuma outra escrita em *X* ocorrer entre os dois acessos.
3. As escritas no mesmo local são *serializadas*, ou seja, duas escritas ao mesmo local por dois processadores quaisquer são vistas na mesma ordem por todos os processadores. Por exemplo, se os valores 1 e depois 2 forem escritos em um local, os processadores não poderão jamais ler o valor do local como 2 e depois como 1.

A primeira propriedade simplesmente preserva a ordem do programa — esperamos que essa propriedade seja verdadeira mesmo em uniprocessadores. A segunda propriedade define a noção do que significa ter uma visão coerente da memória: se um processador pudesse ler continuamente um valor de dados antigo, diríamos que a memória estava incoerente.

A necessidade de serialização de escrita é mais sutil, mas igualmente importante. Suponha que não realizássemos as escritas em série e o processador P1 escrevesse no local *X* seguido por P2 escrevendo no local *X*. A serialização das escritas garante que cada processador verá a escrita feita por P2 no mesmo ponto. Se não realizássemos as escritas em série, algum processador poderia ver primeiro a escrita de P2 e depois a escrita de P1, mantendo o valor escrito por P1 indefinidamente. O modo mais simples de evitar essas dificuldades é garantir que todas as escritas no mesmo local sejam vistas na mesma ordem; essa propriedade é chamada *serialização de escrita*.

Embora as três propriedades recém-descritas sejam suficientes para garantir a coerência, a questão de quanto um valor escrito será visto também é importante. Para ver por que, observe que não podemos exigir que uma leitura de *X* veja instantaneamente o valor escrito para *X* por algum outro processador. Se, por exemplo, uma escrita de X em um processador preceder uma leitura de X em outro processador por um tempo muito pequeno, talvez seja impossível garantir que a leitura retorne o valor dos dados escritos, pois os dados escritos podem nem sequer ter deixado o processador nesse ponto. A questão de exatamente *quando*

um valor de escrita deve ser visto por um leitor é definida por um *modelo de consistência de memória* — um tópico discutido na Seção 5.6.

Coerência e consistência são complementares: a coerência define o comportamento de leituras e escritas no mesmo local da memória, enquanto a consistência define o comportamento de leituras e escritas com relação aos acessos a outros locais da memória. Por enquanto, considere as duas suposições a seguir: 1) uma escrita não termina (e permite que a escrita seguinte ocorra) até que todos os processadores tenham visto o efeito dessa escrita; 2) o processador não muda a ordem de qualquer escrita com relação a qualquer outro acesso à memória. Essas duas condições significam que, se um processador escreve no local *A* seguido pelo local *B*, qualquer processador que vê o novo valor de *B* também precisa ver o novo valor de *A*. Essas restrições permitem que o processador reordene as leituras, mas forçam o processador a terminar uma escrita na ordem do programa. Contaremos com essa suposição até chegarmos à Seção 5.6, na qual veremos exatamente as implicações dessa definição, além das alternativas.

Esquemas básicos para impor a coerência

O problema de coerência para multiprocessadores e E/S, embora semelhante na origem, possui diferentes características que afetam a solução apropriada. Ao contrário da E/S, em que múltiplas cópias de dados são um evento raro — a ser evitado sempre que possível —, um programa que executa em múltiplos processadores normalmente terá cópias dos mesmos dados em várias caches. Em um multiprocessador coerente, as caches oferecem *migração* e *replicação* dos itens de dados compartilhados.

Caches coerentes oferecem migração, pois um item de dados pode ser movido para uma cache local e usado ali em um padrão transparente. Essa migração reduz tanto a latência para acessar um item de dados compartilhado alocado remotamente quanto a demanda de largura de banda na memória compartilhada.

Caches coerentes também oferecem replicação para dados compartilhados que estão sendo lidos simultaneamente, pois as caches criam uma cópia do item de dados na cache local. A replicação reduz tanto a latência de acesso quanto a disputa por um item de dados de leitura compartilhada. O suporte para essa migração e replicação é fundamental para o desempenho no acesso aos dados compartilhados. Assim, em vez de tentar solucionar o problema evitando-o no software, multiprocessadores em pequena escala adotam uma solução de hardware, introduzindo um protocolo para manter caches coerentes.

Os protocolos para manter coerência para múltiplos processadores são chamados *protocolos de coerência de cache*. A chave para implementar um protocolo de coerência de cache é rastrear o estado de qualquer compartilhamento de um bloco de dados. Existem duas classes de protocolos em uso, que empregam diferentes técnicas para rastrear o *estado* de compartilhamento:

- *Baseado em diretório.* O *estado* de compartilhamento de um bloco de memória física é mantido em apenas um local, chamado *diretório.* Existem dois tipos muito diferentes de coerência de cache baseada em diretório. Em um SMP, podemos usar um diretório centralizado, associado à memória ou a algum outro tipo de ponto único de centralização, como a cache mais externa de um multicore. Em um DSM não faz sentido ter um diretório único, uma vez que isso criaria um único ponto de contenção e tornar difícil escalar para muitos chips multicore, dadas as demandas de memória de multicores com oito ou mais núcleos. Diretórios distribuídos são mais complexos do que um diretório único, e tais projetos serão assunto da Seção 5.4.
- *Snooping.* Em vez de manter o estado do compartilhamento em um único diretório, cada cache que tem uma cópia dos dados de um bloco da memória física pode

rastrear o estado de compartilhamento do bloco. Em um SMP, geralmente as caches são acessíveis por meio de algum meio de broadcast (p. ex., um barramento que conecta as caches por núcleo à cache ou à memória compartilhada) e todos os controladores de cache monitoram ou *bisbilhotam* (*snoop*) o meio para determinar se elas têm uma cópia de um bloco que é solicitado em um acesso ao barramento ou a um switch. O snooping também pode ser usado como protocolo de coerência para um multiprocessador multichip, e alguns projetos suportam um protocolo de snooping no topo de um protocolo de diretório dentro de cada multicore!

Os protocolos snooping tornaram-se populares com multiprocessadores que usam microprocessadores (núcleo único) e caches anexados a uma única memória compartilhada por um barramento. Esse barramento proporcionava um meio de transmissão conveniente para implementar os protocolos de snooping. As arquiteturas multicore mudaram significativamente a situação, uma vez que todos os multicores compartilham algum nível de cache no chip. Assim, alguns projetos mudaram para usar protocolos de diretório, uma vez que o overhead era pequeno. Para permitir ao leitor se familiarizar com os dois tipos de protocolo, nos concentraremos em um protocolo snooping e discutiremos um protocolo de diretório quando chegarmos às arquiteturas DSM.

Protocolos de coerência por snooping

Existem duas maneiras de manter o requisito de coerência descrito na subseção anterior. Uma delas é garantir que um processador tenha acesso exclusivo a um item de dados antes que escreva nesse item. Esse estilo de protocolo é chamado *protocolo de invalidação de escrita*, pois invalida outras cópias em uma escrita. De longe, é o protocolo mais comum para os esquemas de snooping e de diretório. O acesso exclusivo garante que nenhuma outra cópia de um item que possa ser lida ou escrita existirá quando houver a escrita: todas as outras cópias do item na cache serão invalidadas.

A Figura 5.4 mostra um exemplo de protocolo de invalidação para um barramento de snooping com caches write-back em ação. Para ver como esse protocolo garante a coerência, considere

Atividade do processador	Atividade do barramento	Conteúdo da cache do processador *A*	Conteúdo da cache do processador *B*	Conteúdo da localização de memória *X*
				0
Processador *A* lê *X*	Cache miss para *X*	0		0
Processador *B* lê *X*	Cache miss para *X*	0	0	0
Processador *A* escreve 1 em *X*	Invalidação para *X*	1		0
Processador *B* lê *X*	Cache miss para *X*	1	1	1

FIGURA 5.4 Exemplo de um protocolo de invalidação que trabalha em barramento de snooping para um único bloco de cache (*X*) com caches write-back.

Consideramos que nenhuma cache mantém inicialmente *X* e que o valor de *X* na memória é 0. O processador e o conteúdo da memória mostram o valor depois que as atividades do processador e do barramento tiverem sido concluídas. Um espaço em branco indica nenhuma atividade ou nenhuma cópia na cache. Quando ocorrer uma segunda falta por *B*, o processador *A* responderá com o valor cancelando a resposta da memória. Além disso, tanto o conteúdo da cache de *B* quanto o conteúdo da memória de *X* são atualizados. Essa atualização da memória, que ocorre quando um bloco se torna compartilhado, simplifica o protocolo, mas só será possível rastrear a propriedade e forçar a escrita de volta se o bloco for substituído. Isso exige a introdução de um estado adicional, chamado "proprietário", que indica que um bloco pode ser compartilhado, mas o processador que o possui é responsável por atualizar quaisquer outros processadores e memória quando alterar o bloco ou substituí-lo. Se um multicore usa uma cache compartilhada (p. ex., L3), toda a memória é vista através da cache compartilhada. L3 age como a memória nesse exemplo, e a coerência deve ser tratada para os L1 e L2 provados de cada núcleo. Foi essa observação que levou alguns projetistas a optarem por um protocolo de diretório dentro do multicore. Para fazer isso funcionar, a cache L3 deve ser inclusiva (página 348).

uma escrita seguida por uma leitura por outro processador: como a escrita exige acesso exclusivo, qualquer cópia mantida pelo processador que está lendo precisa ser invalidada (daí o nome do protocolo). Assim, quando ocorre a leitura, ocorre um miss na cache e ela é forçada a buscar uma nova cópia dos dados. Para uma escrita, exigimos que o processador que está escrevendo tenha acesso exclusivo, para evitar que qualquer outro processador seja capaz de escrever simultaneamente. Se dois processadores tentarem escrever os mesmos dados simultaneamente, um deles vencerá a corrida (veremos como decidir quem vence em breve), fazendo com que a cópia do outro processador seja invalidada. Para que o outro processador complete sua escrita, ele precisa obter uma nova cópia dos dados, que agora tem de conter o valor atualizado. Portanto, esse protocolo impõe a serialização da escrita.

A alternativa a um protocolo de invalidação é atualizar todas as cópias na cache de um item de dados quando esse item é escrito. Esse tipo de protocolo é chamado protocolo de *atualização de escrita* ou *broadcast de escrita*. Como um protocolo de atualização de escrita precisa transmitir por broadcast todas as escritas nas linhas da cache compartilhada, consome muito mais largura de banda. Por esse motivo, todos os multiprocessadores recentes optaram por implementar um protocolo de invalidação de escrita; no restante deste capítulo enfocaremos apenas os protocolos de invalidação.

Técnicas básicas de implementação

A chave para a implementação de um protocolo de invalidação em um multiprocessador em pequena escala é o uso do barramento ou de outro meio de broadcast para realizar as invalidações. Em multiprocessadores de múltiplos chips mais antigos, o barramento usado para coerência era o barramento de acesso à memória compartilhada. Em um multicore, o barramento pode ser a conexão entre as caches privadas (L1 e L2 no Intel Core i7) e a cache externa compartilhada (L3 no i7). Para realizar uma invalidação, o processador simplesmente adquire acesso ao barramento e transmite o endereço a ser invalidado no barramento por broadcast. Todos os processadores realizam um snoop continuamente no barramento, observando os endereços. Os processadores verificam se o endereço no barramento está em sua cache. Se estiver, os dados correspondentes na cache serão invalidados.

Quando ocorre uma escrita em um bloco que é compartilhado, o processador que está escrevendo precisa adquirir o acesso ao barramento para enviar sua invalidação por broadcast. Se dois processadores tentarem escrever em blocos compartilhados ao mesmo tempo, suas tentativas de enviar uma operação de invalidação por broadcast serão serializadas quando disputarem o barramento. O primeiro processador a obter acesso ao barramento fará com que quaisquer outras cópias do bloco que estiver escrevendo sejam invalidadas. Se os processadores estiverem tentando escrever no mesmo bloco, a serialização imposta pelo barramento também serializará suas escritas. Uma implicação desse esquema é que uma escrita em um item de dados compartilhado não pode realmente ser concluída até obter acesso ao barramento. Todos os esquemas de coerência exigem algum método de serializar os acessos ao mesmo bloco de cache, seja serializando o acesso ao meio de comunicação, seja serializando outra estrutura compartilhada.

Além de invalidar cópias pendentes de um bloco de cache que está sendo escrito, também precisamos localizar um item de dados quando ocorre uma cache miss. Em uma cache write-through, é fácil encontrar o valor recente de um item de dados, pois todos os dados escritos são enviados à memória, na qual o valor mais recente de um item de dados sempre pode ser apanhado. (Os buffers de escrita podem levar a algumas complexidades adicionais, por isso devem efetivamente ser tratados como entradas adicionais de cache.)

Para uma cache write-back, o problema de encontrar os valores de dados mais recentes é mais difícil, pois o valor mais recente de um item de dados pode estar em uma cache privada e

não em uma cache compartilhada ou na memória. Felizmente, as caches write-back podem usar o mesmo esquema de snooping, tanto para cache miss quanto para escritas: cada processador bisbilhota cada endereço colocado no barramento. Se um processador descobrir que possui uma cópia modificada do bloco de cache solicitado, oferecerá esse bloco de cache em resposta à solicitação de leitura e fará com que o acesso à memória (ou L3) seja abortado. A complexidade adicional advém do fato de ser necessário recuperar o bloco de cache de uma cache privada de outro processador (L1 ou L2), o que normalmente demorará mais tempo do que recuperá-lo da L3. Como as caches write-back geram requisitos inferiores para largura de banda de memória, eles podem admitir maior quantidade de processadores mais rápidos, e essa tem sido a técnica escolhida na maioria dos multiprocessadores, apesar da complexidade adicional de manter a coerência. Portanto, examinaremos a implementação da coerência com as caches write-back.

As tags de cache normais podem ser usadas para implementar o processo de snooping, e o bit de validade para cada bloco torna a invalidação fácil de implementar. Faltas de leitura, sejam elas geradas por invalidação, seja por algum outro evento, também são diretas, pois simplesmente contam com a capacidade de snooping. Para escritas, gostaríamos de saber se quaisquer outras cópias do bloco estão na cache porque, se não houver outras cópias na cache, a escrita não precisará ser colocada no barramento em uma cache write-back. Não enviar a escrita reduz tanto o tempo gasto pela escrita quanto a largura de banda exigida.

Para rastrear se um bloco de cache é ou não compartilhado, podemos acrescentar um bit de estado extra associado a cada bloco de cache, assim como temos um bit de validade e um bit de modificação. Acrescentando um bit para indicar se o bloco é compartilhado, podemos decidir se uma escrita precisa gerar invalidação. Quando ocorre escrita em um bloco no estado compartilhado, a cache gera invalidação no barramento e marca o bloco como *exclusivo*. Nenhuma outra invalidação será enviada por esse processador para esse bloco. O processador com a única cópia de um bloco de cache normalmente é chamado *proprietário* (owner) do bloco de cache.

Quando uma invalidação é enviada, o estado do bloco de cache do proprietário é trocado de compartilhado para não compartilhado (ou exclusivo). Se outro processador mais tarde exigir esse bloco de cache, o estado precisará se tornar compartilhado novamente. Como nossa cache snooping também vê quaisquer misses, ele sabe quando o bloco de cache exclusivo foi solicitado por outro processador e o estado deve tornar-se compartilhado.

Cada transação do barramento precisa verificar as tags de endereço de cache, que poderiam interferir com os acessos à cache do processador. Uma maneira de reduzir essa interferência é duplicar as tags. Outra abordagem é usar um diretório na cache L3 compartilhada. O diretório indica se um dado bloco é compartilhado e que núcleos podem ter cópias. Com a informação de diretório, invalidações podem ser direcionadas somente para as caches com cópias do bloco de cache. Isso requer que L3 tenha sempre uma cópia de qualquer item de dados em L1 ou L2, uma propriedade chamada *inclusão*, a qual retomaremos na Seção 5.7.

Um exemplo de protocolo

Um protocolo de coerência snooping normalmente é implementado pela incorporação de um controlador de estados finitos em cada nó. Esse controlador responde a solicitações do processador e do barramento (ou de outro meio de broadcast), alterando o estado do bloco de cache selecionado e também usando o barramento para acessar os dados ou invalidá-los. Logicamente, você pode pensar em um controlador separado estando associado a cada bloco, ou seja, as operações de snooping ou solicitações de cache para diferentes blocos podem prosseguir independentemente. Nas implementações reais, um único controlador permite que múltiplas operações para blocos distintos prossigam de

um modo intercalado (ou seja, uma operação pode ser iniciada antes que outra seja concluída, embora somente um acesso à cache ou um acesso ao barramento seja permitido de cada vez). Além disso, lembre-se de que, embora estejamos nos referindo a um barramento na descrição a seguir, qualquer rede de interconexão que admita um broadcast a todos os controladores de coerência e suas caches associadas poderá ser usada para implementar o snooping.

O protocolo simples que consideramos possui três estados: inválido, compartilhado e modificado. O estado compartilhado indica que o bloco é potencialmente compartilhado, enquanto o estado modificado indica que o bloco foi atualizado na cache; observe que o estado modificado *implica* que o bloco é exclusivo. A Figura 5.5 mostra as solicitações geradas pelo módulo de cache do processador em um nó (na metade superior da tabela), além daquelas que vêm do barramento (na metade inferior da tabela). Esse protocolo é para uma cache write-back, mas é facilmente alterado para trabalhar para uma cache write-through, reinterpretando o estado modificado como um estado exclusivo e atualizando a cache nas escritas no padrão normal para uma cache write-through. A extensão mais comum desse protocolo básico é o acréscimo de um estado exclusivo, que descreve um bloco que não é modificado, mas mantido em apenas uma cache privada. Descreveremos essa e outras extensões nas páginas 317 e 318.

Quando uma invalidação ou uma falta de escrita é colocada no barramento, quaisquer núcleos cujas caches privadas têm cópias do bloco de cache a invalidam. Para falta de escrita em uma cache write-back, se o bloco for exclusivo em apenas uma cache, essa cache também escreverá de volta no bloco; caso contrário, os dados podem ser lidos da cache compartilhada ou memória.

A Figura 5.6 mostra um diagrama de transição de estados finitos para um único bloco de cache que usa um protocolo de invalidação de escrita e uma cache write-back. Para simplificar, os três estados do protocolo são duplicados para representar transições com base nas solicitações do processador (à esquerda, que corresponde à metade superior da tabela na Figura 5.5), ao contrário das transições baseadas nas solicitações de barramento (à direita, que corresponde à metade inferior da tabela na Figura 5.5). O texto em negrito é usado para distinguir as ações do barramento, ao contrário das condições em que uma transição de estado depende. O estado em cada nó representa o estado do bloco de cache selecionado, especificado pela solicitação de processador ou de barramento.

Todos os estados nesse protocolo de cache seriam necessários em uma cache de uniprocessador, onde corresponderiam aos estados inválido, válido (e limpo) e modificado. A maioria das mudanças de estados indicadas pelos arcos na metade esquerda da Figura 5.6 seria necessária em uma cache de uniprocessador write-back, exceto a invalidação em um acerto de escrita para um bloco compartilhado. As mudanças de estados representadas pelos arcos na metade direita da Figura 5.6 são necessárias apenas por coerência, e não apareceriam de forma alguma em um controlador de cache de uniprocessador.

Conforme mencionamos, existe apenas uma máquina de estados finitos por cache, com estímulos vindos do processador associado ou do barramento. A Figura 5.7 mostra como as transições de estados na metade direita da Figura 5.6 são combinadas com as da metade esquerda da figura para formar um único diagrama de estados para cada bloco de cache.

Para entender por que esse protocolo funciona, observe que qualquer bloco de cache válido está no estado compartilhado em uma ou mais caches ou no estado exclusivo, exatamente em uma cache. Qualquer transição para o estado exclusivo (que é exigido para que um processador escreva no bloco) exige que uma invalidação ou falta de escrita seja colocada no barramento, fazendo com que todas as caches locais tornem o bloco inválido. Além

Solicitação	Origem	Estado do endereçamento do bloco de cache	Tipo de ação de cache	Função e explicação
Acerto (hit) de leitura	Processador	Compartilhado ou modificado	Acerto (hit) normal	Ler dados na cache.
Falta (miss) de leitura	Processador	Inválido	Falta (miss) normal	Colocar falta de leitura no barramento.
Falta de leitura	Processador	Compartilhado	Substituição	Falta de conflito de endereço: coloca falta de leitura no barramento.
Falta de leitura	Processador	Modificado	Substituição	Falta de conflito de endereço: faz write-back do bloco e depois coloca miss de leitura no barramento.
Acerto(hit) de escrita	Processador	Modificado	Acerto normal	Escrever dados na cache.
Acerto de escrita	Processador	Compartilhado	Coerência	Colocar invalidação no barramento. Normalmente essas operações são chamadas *atualização* ou miss de *propriedade*, pois não buscam os dados, apenas alteram o estado.
Falta de escrita	Processador	Inválido	Falta normal	Colocar falta de escrita no barramento.
Falta de escrita	Processador	Compartilhado	Substituição	Falta de conflito de endereço: coloca falta de escrita no barramento.
Falta de escrita	Processador	Modificado	Substituição	Falta de conflito de endereço: write-back do bloco, depois coloca miss de escrita no barramento.
Falta de leitura	Barramento	Compartilhado	Nenhuma ação	Permitir que a memória atenda a falta de leitura.
Falta de leitura	Barramento	Modificado	Coerência	Tentar compartilhar dados: coloca bloco de cache no barramento e altera estado para compartilhado.
Invalidação	Barramento	Compartilhado	Coerência	Tentar escrever em bloco compartilhado; invalida o bloco.
Falta de escrita	Barramento	Compartilhado	Coerência	Tentar escrever em bloco compartilhado; invalida o bloco de cache.
Falta de escrita	Barramento	Modificado	Coerência	Tentar escrever em bloco que é exclusivo de outro lugar: escreve o bloco de cache de volta e torna seu estado inválido.

FIGURA 5.5 O mecanismo de coerência de cache recebe solicitações tanto do processador quanto do barramento e as responde com base no tipo de solicitação, se ela acerta ou falta na cache local, e o estado do bloco de cache especificado na solicitação.
A quarta coluna descreve o tipo de ação de cache como acerto ou falta normal (o mesmo que uma cache de uniprocessador veria), substituição (uma falta de substituição, uma falta de cache do uniprocessador) ou coerência (exigida para manter a coerência da cache); uma ação normal ou de substituição pode causar uma ação de coerência, dependendo do estado do bloco em outras caches. Para falta de leitura, faltas, faltas de escrita ou invalidações monitoradas do barramento, uma ação é necessária *somente* se os endereços de leitura ou escrita corresponderem a um bloco na cache e o bloco for válido.

disso, se alguma outra cache local tiver o bloco no estado exclusivo, essa cache local gera um write-back, que fornece o bloco com o endereço desejado. Finalmente, se houver falta de leitura no barramento para um bloco no estado exclusivo, a cache local com a cópia exclusiva mudará seu estado para compartilhado.

As ações em cinza na Figura 5.7, que tratam de miss de leitura e de escrita no barramento, são essencialmente o componente snooping do protocolo. Outra propriedade que é preservada nesse e na maioria dos outros protocolos é que qualquer bloco de memória no estado compartilhado sempre está atualizado em relação à cache externa compartilhada (L2 ou L3 ou memória, e não existir cache compartilhada), o que simplifica a implementação.

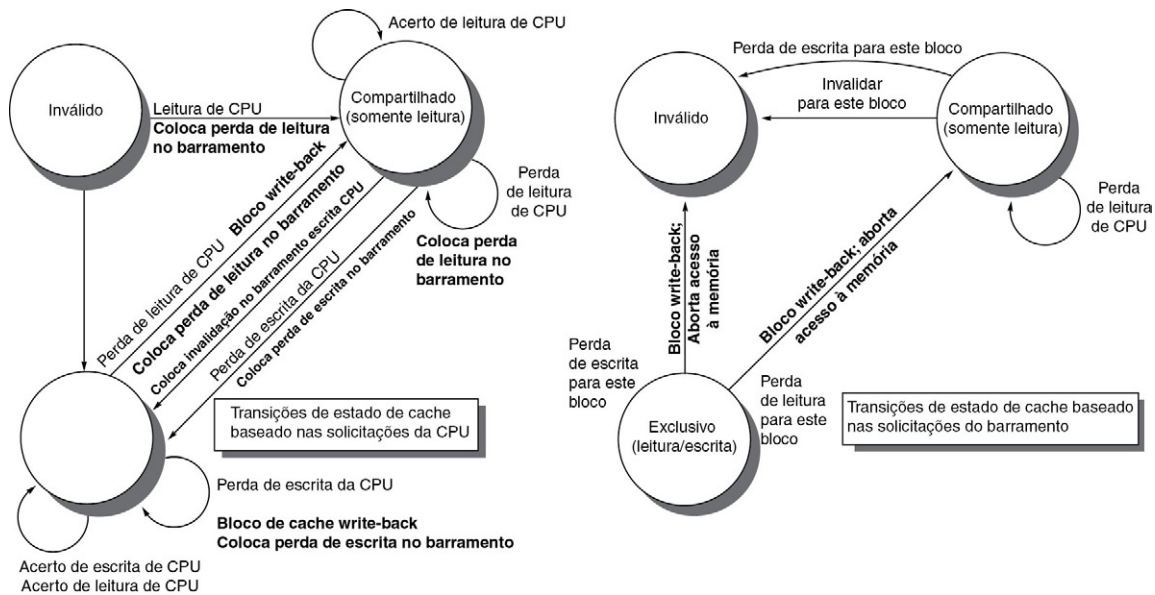



FIGURA 5.6 **Protocolo de invalidação de escrita, coerência de cache, para uma cache privada write-back, mostrando os estados e as transições de estados para cada bloco na cache.**

Os estados da cache aparecem em círculos, com qualquer acesso permitido pelo processador local sem transição de estado sendo mostrado entre parênteses, sob o nome do estado. O estímulo que causa uma mudança de estado aparece nos arcos de transição em tipo normal, e quaisquer ações do barramento geradas como parte da transição de estado aparecem no arco de transição em negrito. As ações de estímulo se aplicam a um bloco na cache, e não a um endereço específico na cache. Logo, uma falta de leitura para um bloco no estado compartilhado é um miss para esse bloco de cache, mas para um endereço diferente. O lado esquerdo do diagrama mostra as transições de estado com base nas ações do processador associados a essa cache; o lado direito mostra as transições com base nas operações sobre o barramento. Falta de leitura no estado exclusivo ou compartilhado e falta de escrita no estado exclusivo ocorrem quando o endereço solicitado pelo processador não combina com o endereço no bloco de cache. Tal falta é uma falta de substituição de cache-padrão. Uma tentativa de escrever um bloco no estado compartilhado gera invalidação. Sempre que ocorre uma transação no barramento, todas as caches que contêm o bloco de cache especificado na transação do barramento tomam a ação indicada pela metade direita do diagrama. O protocolo considera que a memória (ou a cache compartilhada) oferece dados em uma falta de leitura para um bloco que é limpo em todas as caches. Nas implementações reais, esses dois conjuntos de diagramas de estado são combinados. Na prática, existem muitas variações sutis nos protocolos de invalidação, incluindo a introdução do estado não modificado exclusivo quanto ao fato de um processador ou memória oferecer dados em uma falta. Em um chip multicore, a cache compartilhada (geralmente L3, mas às vezes L2) age como o equivalente da memória, e o barramento é o barramento entre as caches privadas de cada núcleo e a cache compartilhada, que, por sua vez, tem interfaces com a memória.

Na verdade, não importa se o nível fora das caches privadas é uma cache ou memória compartilhada. A chave é que todos os acessos dos núcleos passam por esse nível.

Embora nosso protocolo de cache simples esteja correto, ele omite uma série de complicações que tornam a implementação muito mais complicada. A mais importante delas é que o protocolo considera que as operações são *atômicas* — ou seja, uma operação pode ser feita de modo que nenhuma operação intermediária possa ocorrer. Por exemplo, o protocolo descrito considera que as faltas de escrita podem ser detectadas, que o barramento pode ser tomado e que uma resposta possa ser dada como uma única ação indivisível. Na realidade, isso não é verdade. De fato, mesmo um miss de leitura pode não ser indivisível. Depois de detectar um miss no L2 de um multicore, o núcleo deve decidir entre acessar o barramento que o conecta à cache compartilhada L3. Ações não indivisíveis introduzem a possibilidade de o protocolo sofrer *deadlock*, significando que ele chega a um estado em que não pode continuar. Em breve, exploraremos essas complicações nesta seção, quando examinarmos projetos DSM.

Com os processadores multicore, a coerência entre os núcleos do processador é toda implementada no chip, usando um protocolo snooping ou diretório central simples. Muitos chips com dois processadores, incluindo o Intel Xeon e AMD Opteron, suportavam

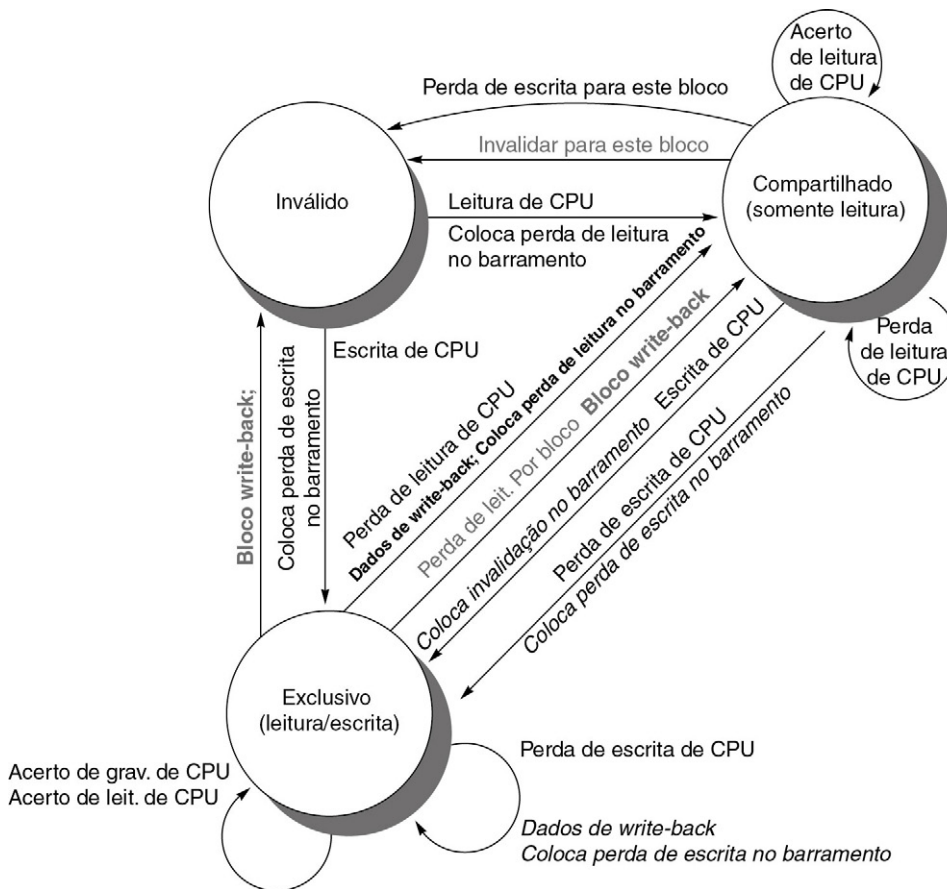



FIGURA 5.7 Diagrama de estado de coerência de cache com as transições de estado induzidas pelo processador local, mostradas em preto, e pelas atividades de barramento, mostradas em cinza. Assim como na Figura 5.6, as atividades em uma transição aparecem em negrito.

multiprocessadores com múltiplos chips que poderiam ser construídos conectando uma interface de alta velocidade (chamadas Quickpath ou Hypertranspor, respectivamente). Esses tipos de interconexões não são apenas extensões do barramento compartilhado, mas usam uma abordagem diferente para multicores interconectados.

Um multiprocessador construído com múltiplos chips multicore terá uma arquitetura de memória compartilhada e precisará de um mecanismo de coerência interna ao chip, acima e além daquela existente dentro do chip. Na maioria dos casos, alguma forma de esquema de diretório é usada.

Extensões do protocolo básico de coerência

O protocolo de coerência que acabamos de descrever é um simples protocolo de três estados e, muitas vezes, é chamado pela primeira letra dos estados, fazendo dele um protocolo MSI (Modificado, Compartilhado, Inválido — *Modified, Shared, Invalid*). Existem muitas extensões para esse protocolo básico, que mencionamos nas legendas desta seção. Essas extensões são criadas pela adição de estados e transações, que otimizam certos comportamentos, possivelmente resultando em melhor desempenho. Duas das extensões mais comuns são:

1. MESI adiciona o estado Exclusivo ao protocolo básico MSI para indicar quando um bloco de cache é residente somente em uma cache única, mas está limpo.

Se um bloco estiver no estado E, ele pode ser gravado sem gerar nenhuma invalidação, o que otimiza o caso em que um bloco é lido por uma única cache antes de ser escrito por ele. Obviamente, quando um miss de leitura para um bloco no estado E ocorre, o bloco deve ser modificado para o estado S para manter a coerência. Uma vez que todos os acessos subsequentes são monitorados, é possível manter a coerência desse estado. Em particular, se outro processador despachar um miss de leitura, o estado é mudado de exclusivo para compartilhado. A vantagem de adicionar esse estado é que uma gravação subsequente para um bloco no estado exclusivo pelo mesmo núcleo não precisa obter acesso ao barramento ou gerar uma invalidação, já que se sabe que o bloco está exclusivamente nessa cache local. O processador simplesmente muda o estado para modificado. Esse estado é adicionado facilmente usando o bit que codifica o estado coerente como um estado exclusivo e usando o bit modificado para indicar que um bloco foi modificado. O popular protocolo MESI, que recebe o nome dos quatro estados que ele inclui (Modificado, Exclusivo, Compartilhado e Inválido), usa essa estrutura. O Intel i7 usa uma variação de um protocolo MESI, chamada MESIF, que adiciona um estado (*Forward*) para designar que o processador que está compartilhando deve responder a uma requisição. Isso é projetado para aumentar o desempenho em organizações de memória distribuída.

2. MOESI adiciona o estado *Owned* (*proprietário*) para o protocolo MESI para indicar que o bloco associado é de propriedade daquela cache e está desatualizado na memória. Em protocolos MSI e MESI, quando há uma tentativa de compartilhar um bloco no estado Modificado, o estado é mudado para Compartilhado (tanto na cache original quanto na que agora está compartilhada), e o bloco deve ser escrito de volta na memória. Em um protocolo MOESI, o bloco pode ser mudado do estado Modificado para o estado Owned na cache original sem gravá-lo na memória. Outras caches, que agora estão compartilhando o bloco, mantêm o bloco no estado Compartilhado. O estado O, que somente a cache original mantém, indica que a cópia na memória principal está desatualizada e que a cache designada é a proprietária. A proprietária do bloco deve fornecê-lo no caso de uma falta, já que a memória não está atualizada e deve gravar o bloco de volta na memória se ele for substituído. O AMD Opteron usa o protocolo MOESI.

A próxima seção examinará o desempenho desses protocolos para nossas cargas de trabalho paralelas e multiprogramadas. O valor dessas extensões para um protocolo básico ficará claro quando examinarmos o desempenho. Mas, antes de fazermos isso, vamos dar uma rápida olhada nas limitações no uso de uma estrutura simétrica de memória e um esquema de coerência snooping.

Limitações nos multiprocessadores simétricos de memória compartilhada e protocolos de snooping

À medida que o número de processadores em um multiprocessador cresce ou as demandas de memória de cada processador aumentam, qualquer recurso centralizado no sistema pode se tornar um gargalo. Usando a maior conexão de largura de banda disponível no chip e uma cache L3 compartilhada, que é mais rápido do que a memória, os projetistas têm conseguido suportar quatro a oito núcleos de alto desempenho de modo simétrico. Tal abordagem provavelmente não vai muito além de oito núcleos, e não vai funcionar quando múltiplos multicores forem combinados.

A largura de banda de snooping nas caches também pode se tornar um problema, já que cada cache deve examinar cada falta colocada no barramento. Como mencionamos, duplicar as tags é uma solução. Outra abordagem, que tem sido adotada em alguns

multicores recentes, é colocar um diretório no nível da cache mais externa. O diretório indica explicitamente as caches em que o processador tem cópias de cada item da cache mais externa. Essa é a abordagem que a Intel usa nas séries i7 e Xeon 7000. Observe que o uso desse diretório não elimina o gargalo devido a um barramento e uma L3 compartilhados entre os processadores, mas é muito mais simples de implementar do que os esquemas de diretório distribuído que vamos examinar na Seção 5.4.

Como um projetista poderia aumentar a largura de banda da memória para dar suporte a mais processadores e a processadores mais rápidos? Para aumentar a largura de banda de comunicação entre os processadores e a memória, os projetistas têm usado vários barramentos e também redes de interconexão, como crossbars ou pequenas redes ponto a ponto. Nesses projetos, o sistema de memória pode ser configurado em múltiplos bancos físicos, de modo a aumentar a largura de banda efetiva da memória enquanto retém o tempo de acesso uniforme à memória. A Figura 5.8 mostra essa técnica, que representa um ponto intermediário entre as duas técnicas que discutimos no início do capítulo: memória compartilhada centralizada e memória compartilhada distribuída.

O AMD Opteron representa outro ponto intermediário no espectro entre um protocolo snooping e de diretório. A memória está conectada diretamente a cada chip multicore, e até quatro chips multicore podem estar conectados. O sistema é NUMA, já que a memória local é um pouco mais rápida. O Opteron implementa seu protocolo de coerência usando os links ponto a ponto para transmitir por broadcast a até três outros chips. Como os links entre os processadores não são compartilhados, a única maneira de um processador saber quando uma operação inválida foi concluída é uma confirmação explícita. Assim, o protocolo de coerência utiliza um broadcast para encontrar cópias potencialmente

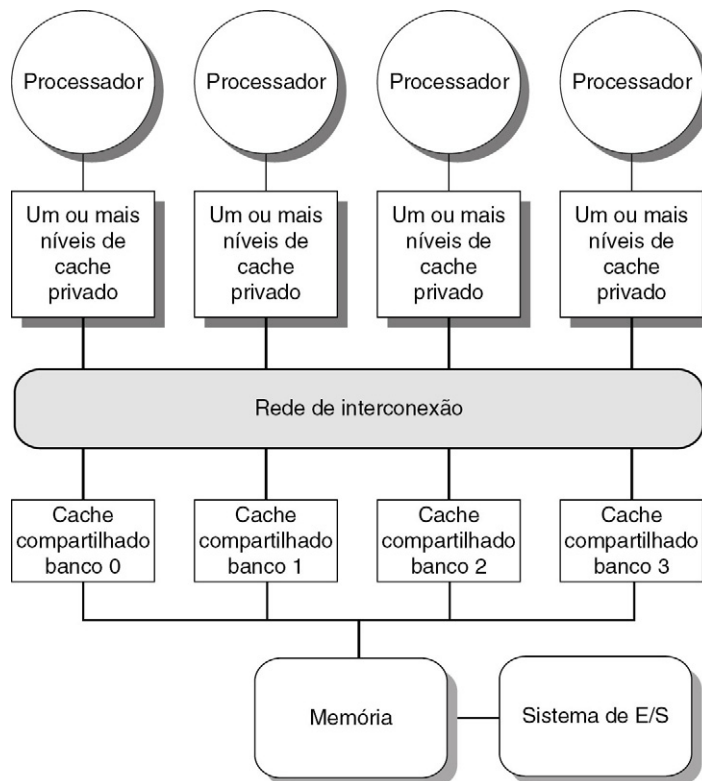



FIGURA 5.8 Multiprocessador com acesso uniforme à memória que usa bancos cache compartilhada e rede de interconexão em vez de barramento.

compartilhadas, como um protocolo snooping, mas usa as confirmações para ordenar as operações, como um protocolo de diretório. Já que a memória local é só um pouco mais rápida do que a memória remota na implementação Opteron, alguns softwares tratam um multiprocessador Opteron como tendo acesso uniforme à memória.

Um protocolo de coerência de cache snooping pode ser usado sem barramento centralizado, mas ainda exigir que um broadcast seja feito para monitorar as caches individuais em cada falta em um potencial bloco de cache compartilhado. Esse tráfego de coerência de cache cria outro limite na escala e na velocidade dos processadores. Como o tráfego de coerência não é afetado por caches maiores, os processadores mais rápidos inevitavelmente sobrecarregarão a rede e a capacidade de cada cache responderá a solicitações de snoop de *todas* as outras caches. Na Seção 5.4, examinaremos os protocolos baseados em diretório, que eliminam a necessidade de broadcast para todas as outras caches, em uma falta. À medida que aumentam as velocidades de processador e o número de núcleos por processador, mais projetistas provavelmente optam por tais protocolos para evitar o limite de broadcast de um protocolo snooping.

Implementando coerência snooping de cache

O diabo está nos detalhes.

Provérbio clássico

Quando escrevemos a primeira edição deste livro, em 1990, nossa seção final "Juntando tudo" foi um multiprocessador de 30 processadores e único barramento, usando a coerência baseada em snoop; o barramento tinha uma capacidade pouco acima de 50 MB/s, que em 2011 não seria largura de banda de barramento suficiente para dar suporte nem mesmo a um Intel i7! Quando escrevemos a segunda edição deste livro, em 1995, os primeiros multiprocessadores de coerência de cache com mais de um barramento tinham aparecido recentemente; acrescentamos um apêndice para descrever a implementação do snooping em um sistema com múltiplos barramentos. Em 2011, a maioria dos processadores multicore que suportavam somente um multiprocessador de chip único optou por usar uma estrutura de barramento compartilhado conectada a uma memória compartilhada ou a uma cache compartilhada. Em contraste, *todos* os sistemas multiprocessadores multicore que suportam 16 ou mais núcleos não utilizam uma interconexão de único barramento, e os projetistas precisam encarar o desafio de implementar o snooping sem a simplificação de um barramento para colocar os eventos em série.

Como já dissemos, a principal complicação para implementar o protocolo de coerência snooping que descrevemos é que as faltas de escrita e de atualização não são indivisíveis em qualquer multiprocessador recente. As etapas de detecção de uma falta de escrita ou de atualização, comunicação com outros processadores e com a memória, obtenção do valor mais recente para uma falta de escrita e garantia de que quaisquer invalidações são processadas, e a atualização da cache não pode ser feita como se utilizassem um único ciclo.

Em um único chip multicore, essas etapas podem se tornar efetivamente indivisíveis apanhando o barramento primeiro (antes de alterar o estado da cache) e não liberando o barramento até que todas as ações sejam concluídas. Como o processador pode saber quando todas as invalidações foram concluídas? Em alguns multicore, uma única linha é usada para sinalizar quando todas as invalidações necessárias foram recebidas e estão sendo processadas. Após esse sinal, o processador que gerou a falta pode liberar o barramento sabendo que quaisquer ações exigidas serão concluídas antes de qualquer atividade relacionada à próxima falta. Mantendo o barramento exclusivamente durante essas etapas, o processador efetivamente torna as etapas individuais indivisíveis.

Em um sistema sem barramento, temos que encontrar algum outro método para tornar as etapas indivisíveis em caso de falta. Em particular, temos que garantir que dois processadores que tentam escrever no mesmo bloco ao mesmo tempo, uma situação chamada *corrida*, sejam estritamente ordenados: uma escrita é processada e prossegue antes que a próxima seja iniciada. Não importa qual das duas escritas vença a corrida, apenas que haja uma única vencedora, cujas ações de coerência sejam completadas primeiro. Em um sistema snooping, fazer com que uma corrida tenha apenas um vencedor é algo garantido pelo uso do broadcast para todas as faltas, além de algumas propriedades básicas da rede de interconexão. Essas propriedades, junto com a capacidade de reiniciar o tratamento de falta do perdedor de uma corrida, são a chave para implementar a coerência snooping de cache sem barramento. Explicaremos os detalhes no Apêndice I.

É possível combinar snooping e diretórios, e muitos projetistas usam snooping dentro de um multicore e diretórios entre múltiplos chips ou *vice-versa*, diretórios dentro de um multicore e snooping entre múltiplos chips.

5.3 DESEMPENHO DE MULTIPROCESSADORES SIMÉTRICOS DE MEMÓRIA COMPARTILHADA

Em um multiprocessador que usa protocolo de coerência snoopy, diversos fenômenos diferentes são combinados para determinar o desempenho. Em particular, o desempenho geral da cache é uma combinação do comportamento do tráfego de cache miss do uniprocessador e do tráfego causado pela comunicação, que resulta em invalidações e subsequentes cache miss. A mudança da quantidade de processadores, do tamanho da cache e do tamanho do bloco pode afetar esses dois componentes da taxa de falta (miss rate) de diferentes maneiras, levando a um comportamento geral do sistema que é uma combinação dos dois efeitos.

O Apêndice B desmembra a taxa de falta do uniprocessador na classificação dos três *C* (capacity, compulsory e conflict) e oferece *percepções* tanto para o comportamento da aplicação como para possíveis melhorias no projeto da cache. De modo semelhante, as faltas que surgem da comunicação entre processadores, que normalmente são chamadas *faltas de coerência*, podem ser desmembradas em duas origens separadas.

A primeira origem são as chamadas *faltas de compartilhamento verdadeiros*, que surgem da comunicação dos dados pelo mecanismo de coerência de cache. Em um protocolo baseado em invalidação, a primeira escrita por um processador a um bloco de cache compartilhado causa invalidação para estabelecer a posse desse bloco. Além disso, quando outro processador tenta ler uma palavra modificada nesse bloco de cache, ocorre uma falta e o bloco resultante é transferido. Essas duas faltas são classificadas como faltas de compartilhamento verdadeiras, pois surgem diretamente do compartilhamento de dados entre os processadores.

O segundo efeito, chamado *compartilhamento falso*, surge do uso de um algoritmo de coerência baseado em invalidação com um único bit de validade por bloco de cache. O compartilhamento falso ocorre quando um bloco é invalidado (e uma referência subsequente causa uma falta), pois alguma palavra no bloco, fora a que está sendo lida, é escrita. Se a palavra escrita for realmente usada pelo processador que recebeu a invalidação, então a referência foi uma referência de compartilhamento verdadeira e teria causado uma falta, independentemente do tamanho do bloco. Porém, se a palavra que está sendo escrita e a palavra lida forem diferentes e a invalidação não fizer com que um novo valor seja comunicado, apenas causando uma falta de cache extra, então ela será uma falta de compartilhamento

falso. Em uma falta de compartilhamento falso, o bloco é compartilhado, mas nenhuma palavra na cache é realmente compartilhada, e a falta não ocorreria se o tamanho do bloco fosse uma única palavra. O exemplo a seguir esclarece os padrões de compartilhamento.

Exemplo Suponha que as palavras x1 e x2 estejam no mesmo bloco de cache, que está no estado compartilhado nas caches de P1 e P2. Considerando a sequência de eventos a seguir, identifique cada falta como uma falta de compartilhamento verdadeiro, uma falta de compartilhamento falso ou um acerto. Qualquer falta que ocorrer, se o tamanho de bloco for de uma palavra, será designada como falta de compartilhamento verdadeira.

Tempo	P1	P2
1	Escreve x1	
2		Lê x2
3	Escreve x1	
4		Escreve x2
5	Lê x2	

Resposta Aqui estão as classificações por etapa no tempo:

1. Esse evento é uma falta de compartilhamento verdadeira, pois x1 foi lido por P2 e precisa ser invalidado de P2.
2. Esse evento é uma falta de compartilhamento falsa, pois x2 foi invalidado pela escrita de x1 em P1, mas esse valor de x1 não é usado em P2.
3. Esse evento é uma falta de compartilhamento falsa, pois o bloco contendo x1 é marcado como compartilhado, devido à leitura em P2, mas P2 não leu x1. O bloco de cache contendo x1 estará no estado compartilhado depois da leitura por P2; uma falta de escrita será necessária para obter acesso exclusivo ao bloco. Em alguns protocolos, isso será tratado como uma solicitação de upgrade, que gera invalidação do barramento, mas não transfere o bloco de cache.
4. Esse evento é uma falta de compartilhamento falsa, pelo mesmo motivo da etapa 3.
5. Esse evento é uma falta de compartilhamento verdadeira, pois o valor sendo lido foi escrito por P2.

Embora vejamos os efeitos das faltas de compartilhamento verdadeiros e falsos nas cargas de trabalho comerciais, o papel das faltas de coerência é mais significativo para aplicações fortemente acopladas, que compartilham quantidades significativas de dados do usuário. Examinaremos seus efeitos com detalhes no Apêndice I, quando considerarmos o desempenho de uma carga de trabalho científica paralela.

Uma carga de trabalho comercial

Nesta seção, examinaremos o comportamento do sistema de memória para um multiprocessador de memória compartilhada com quatro processadores ao rodar uma carga de trabalho comercial de uso geral. O estudo que examinaremos foi feito em 1998, em um sistema Alpha de quatro processadores, mas continua sendo o mais completo e esclarecedor para o desempenho de um multiprocessador para tais cargas de trabalho. Os resultados foram colhidos em um AlphaServer 4100 ou usando um simulador configurável modelado no AlphaServer 4100. Cada processador no AlphaServer 4100 é um Alpha 21164, que despacha até quatro instruções por clock e trabalha em 300 MHz. Embora a frequência do processador Alpha nesse sistema seja consideravelmente mais lenta do que os processadores nos sistemas projetados em 2011, a estrutura básica do sistema, consistindo em

Nível de cache	Característica	Alpha 21164	Intel i7
L1	Tamanho	8 KB I/8 KB D	32 KB I/32 KB D
	Associatividade	Mapeado diretamente	4 vias I/ 8 vias D
	Tamanho de bloco	32 B	64 B
	Penalidade de falta	7	10
L2	Tamanho	96 KB	256 KB
	Associatividade	3 vias	8 vias
	Tamanho de bloco	32 B	64 B
	Penalidade de falta	21	35
L3	Tamanho	2 MB	2 MB por núcleo
	Associatividade	Mapeado diretamente	16 vias
	Tamanho de bloco	64 B	64 B
	Penalidade de falta	80	~100

FIGURA 5.9 Características da hierarquia de cache do Alpha 21164 usado neste estudo e no Intel i7.
Embora os tamanhos sejam maiores e a associabilidade seja maior no i7, as penalidades de falta também são maiores, então o comportamento pode diferir muito pouco. Por exemplo, do Apêndice B, podemos estimar as taxas de falta da cache L1 menor do Alpha como 4,9% e 3% para a cache L1 maior do i7, então a penalidade de falta média de L1 por referência é 0,34 para o Alpha e 0,30 para o i7. Os dois sistemas têm uma penalidade alta (125 ciclos ou mais) para uma transferência requerida a partir de uma cache privada. O i7 também compartilha L3 entre todos os núcleos.

um processador de quatro despachos e uma hierarquia de cache de três níveis, é muito similar à do multicore Intel i7 e outros processadores, como mostrado na Figura 5.9. Em particular, as caches do Alpha são um pouco menores, mas os tempos de falta (miss times) também são menores que os de um i7. Assim, o comportamento do sistema Alpha deve fornecer *percepções* interessantes sobre o comportamento dos projetos multicore modernos.

A carga de trabalho usada para esse estudo consiste em três aplicações:

1. Uma carga de trabalho de processamento de transação on-line (OLPT) modelada no TPC-B (que tem comportamento de memória semelhante ao seu primo mais novo, o TPC-C, descrito no Capítulo 1) e usando Oracle 7.3.2 como sistema de banco de dados. A carga de trabalho consiste em um conjunto de processos clientes que geram solicitações e um conjunto de servidores que os tratam.
   Os processos servidores consomem 85% do tempo do usuário, com o restante indo para os clientes. Embora a latência de E/S seja escondida pelo ajuste cuidadoso e por solicitações suficientes para manter a CPU ocupada, os processos servidores normalmente são bloqueados para E/S após cerca de 25.000 instruções.
2. Uma carga de trabalho de um sistema de apoio à decisão (decision support system — DSS) baseada no TPC-D e também usando Oracle 7.3.2 como sistema de banco de dados. A carga de trabalho inclui apenas seis das 17 consultas de leitura no TPC-D, embora as seis consultas examinadas no benchmark se espalhem por toda a gama de atividades do benchmark inteiro. Para encobrir a latência de E/S, o paralelismo é explorado tanto dentro das consultas, em que o paralelismo é detectado durante um processo de formulação de consulta, quanto entre as consultas. As chamadas de bloqueio são muito menos frequentes do que no benchmark OLTP; as seis consultas têm, em média, cerca de 1,5 milhão de instruções antes de bloquear.
3. Um benchmark de busca de índice da Web (AltaVista), baseado em uma busca de uma versão mapeada na memória do banco de dados do AltaVista (200 GB). O loop interno é altamente otimizado. Como a estrutura de busca é estática, pouco sincronismo é necessário entre os threads. O AltaVista era o sistema de busca na Web mais popular antes da chegada do Google.

Benchmark	% tempo no modo usuário	% tempo no kernel	% tempo ocioso do processador
OLTP	71	18	11
DSS (média por todas as consultas)	87	4	9
AltaVista	>98	<1	<1

FIGURA 5.10 **Distribuição do tempo de execução nas cargas de trabalho comerciais.**
O benchmark OLTP tem a maior fração do tempo de SO e tempo ocioso da CPI (que é o tempo de espera de E/S). O benchmark DSS mostra muito menos tempo de SO, pois realiza menos E/S, porém apresenta mais de 9% de tempo ocioso. O ajuste extensivo do mecanismo de busca do AltaVista é evidente nessas medições. Os dados para essa carga de trabalho foram coletados por Barroso, Gharachorloo e Bugnion (1998) em um AlphaServer 4100 de quatro processadores.

As porcentagens de tempo gastas no modo usuário, no kernel e no loop ocioso aparecem na Figura 5.10. A frequência da E/S aumenta tanto o tempo do kernel quanto o tempo ocioso (ver entrada OLTP, que tem a maior razão entre E/S e a computação). O AltaVista, que mapeia o banco de dados de busca inteiro na memória e tem sido extensivamente ajustado, mostra o menor tempo de kernel ou ocioso.

Medições de desempenho da carga de trabalho comercial

Começamos com um exame geral da execução do processador para esses benchmarks no sistema de quatro processadores; conforme discutiremos na página 322, esses benchmarks incluem tempo de E/S substancial, que é ignorado nas medições de tempo do processador. Agrupamos as seis consultas do DSS como um único benchmark, informando o comportamento médio. O CPI efetivo varia muito para esses benchmarks, de um CPI de 1,3 para a busca na Web do AltaVista, até um CPI médio de 1,6 para a carga de trabalho do DSS e 7,0 para a carga de trabalho do OLTP. A Figura 5.11 mostra como o tempo de execução é desmembrado em execução de instrução, tempo de acesso da cache, sistema de memória e outros stalls (que são, sobretudo, stalls de recursos de pipeline, mas também incluem TLB e stalls de erro de previsão de desvio). Embora o desempenho das cargas de trabalho do DSS e do AltaVista seja razoável, o desempenho da carga de trabalho do OLTP é muito fraco, devido a um fraco desempenho da hierarquia de memória.

Como a carga de trabalho do OLTP exige mais do sistema de memória, com grande quantidade dispendiosa de faltas na L3, enfocamos o exame do impacto do tamanho de cache L3, número de processadores e tamanho de bloco no benchmark OLTP. A Figura 5.12 mostra o efeito de aumentar o tamanho da cache, usando caches associativas por conjunto de duas vias, o que reduz o grande número de faltas em conflito. O tempo de execução é melhorado à medida que a cache L3 cresce, devido à redução nas faltas do L3. É surpreendente que quase todo o ganho se dá passando-se de 1 para 2 MB, com pouco ganho adicional além disso, apesar do fato de as faltas de cache ainda serem uma causa de perda de desempenho significativa com caches de 2 MB e 4 MB. A questão é: por quê?

Para entender melhor a resposta a essa pergunta, precisamos determinar que fatores contribuem para a taxa de falta do L3 e como eles mudam à medida que a cache L3 cresce. A Figura 5.13 mostra esses dados ao apresentar o número de ciclos de acesso à memória contribuídos por instrução, a partir de cinco origens. As duas maiores origens de ciclos de acesso à memória L3, com um L3 de 1 MB, são faltas de instrução e capacidade/conflito. Com um L3 maior, essas duas origens encurtam para serem contribuintes menores. Infelizmente, as faltas de compartilhamento compulsórias, falsas, e as faltas de compartilhamento verdadeiras não são afetadas por uma L3 maior. Assim, em 4 MB e em 8 MB,

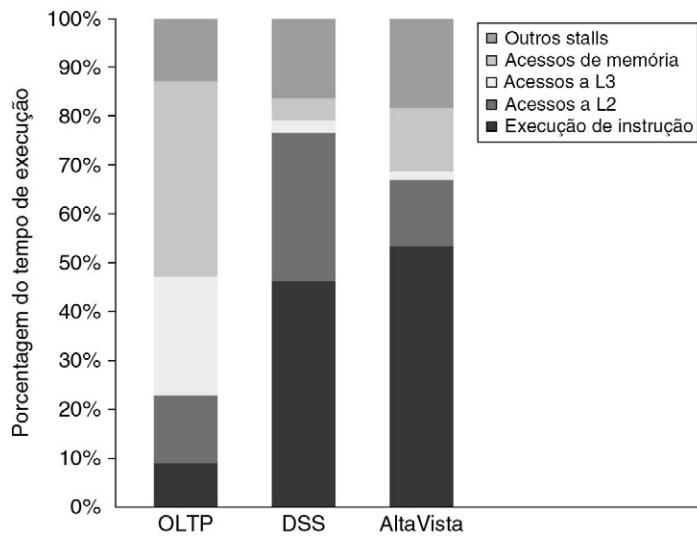



FIGURA 5.11 **Desmembramento do tempo de execução para os três programas (OLTP, DSS e AltaVista) na carga de trabalho comercial.**

Os números do DSS são a média por seis consultas diferentes. O CPI varia muito, desde 1,3 para o AltaVista e 1,61 para as consultas do DSS até 7,0 para o OLTP. (Individualmente, as consultas do DSS mostram um intervalo de CPI de 1,3 a 1,9.) Outros stalls incluem stalls de recursos (implementados com traps de replay no 21164), erro de previsão de desvio, limite de memória e faltas de acesso no TLB. Para esses benchmarks, os stalls de pipeline devido aos seus recursos são fatores dominantes. Esses dados combinam o comportamento do usuário e os acessos ao kernel. Somente o OLTP possui uma fração significativa dos acessos do kernel, e esses acessos costumam ser mais bem comportados do que os acessos do usuário. Todas as medições mostradas nesta seção foram coletadas por Barroso, Gharachorloo e Bugnion (1998).

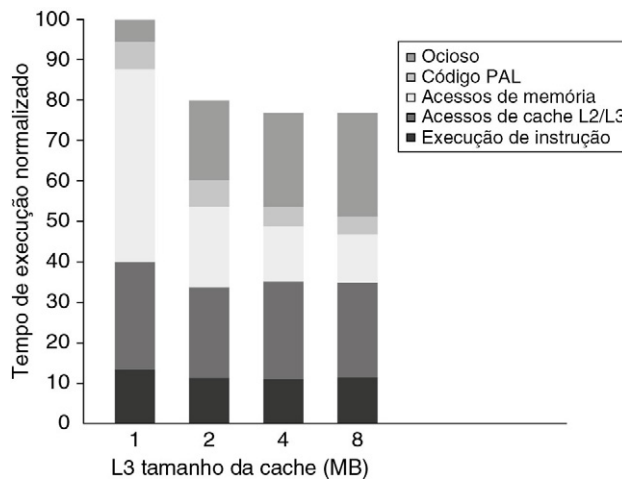



FIGURA 5.12 **Desempenho relativo da carga de trabalho do OLTP à medida que o tamanho da cache L3, que é definido como associativo por conjunto de duas vias, cresce de 1 MB para 8 MB.**

O tempo ocioso também cresce à medida que a cache aumenta, reduzindo alguns dos ganhos de desempenho. Esse crescimento ocorre porque, com menos stalls do sistema de memória, mais processos do servidor são necessários para cobrir a latência de E/S. A carga de trabalho poderia ser retornada para aumentar o equilíbrio de computação/comunicação, mantendo o tempo ocioso sob controle. O código PAL é um conjunto de sequências de instruções especializadas em nível de SO executadas em modo privilegiado. Um exemplo é o tratamento de falta no TLB.

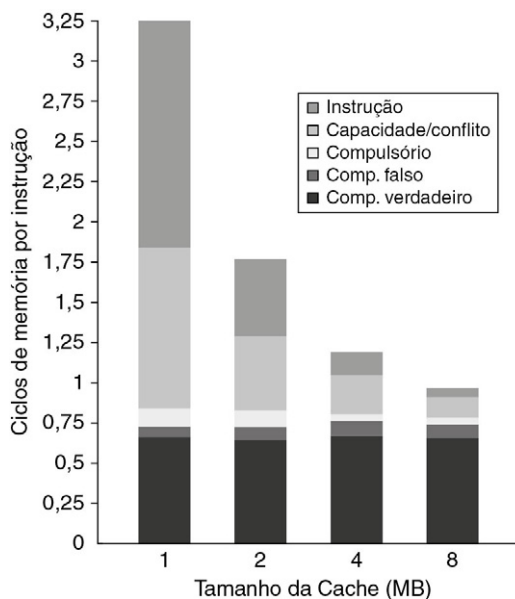



FIGURA 5.13 **Causas contribuintes de mudança de ciclos de acesso à memória à medida que o tamanho da cache aumenta.**
A cache L3 é simulada como associativo por conjunto de duas vias.

as faltas de compartilhamento verdadeiras geram a fração dominante das faltas; a falta de mudança nas faltas de compartilhamento verdadeiras leva a reduções limitadas na taxa de falta geral ao aumentar o tamanho da cache L3 para além de 2 MB.

Aumentar o tamanho da cache elimina a maioria das faltas de um uniprocessador, enquanto deixa as faltas de multiprocessadores intocáveis. Como o aumento na quantidade de processadores afeta diferentes tipos de faltas? A Figura 5.14 mostra esses dados, con-

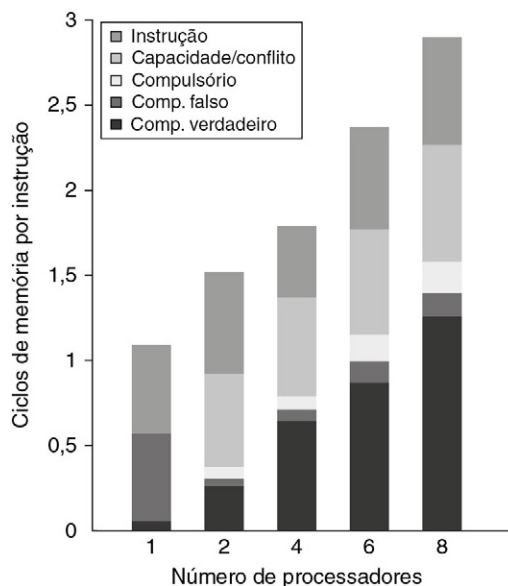



FIGURA 5.14 **A contribuição para os ciclos de acesso à memória aumenta à medida que o número de processadores aumenta, principalmente devido ao aumento do compartilhamento verdadeiro.**
As faltas compulsórias aumentam ligeiramente, pois agora cada processador precisa tratar de mais faltas compulsórias.

siderando uma configuração básica com cache L3 de 2 MB, associativa por conjunto, de duas vias. Como poderíamos esperar, o aumento na taxa de falta de compartilhamento verdadeira, que não é compensado por qualquer diminuição nas faltas de uniprocessador, leva a um aumento geral nos ciclos de acesso à memória por instrução.

A última questão que examinaremos é se o aumento do tamanho de bloco — que deverá diminuir a taxa de falta de instrução e manter a taxa de faltas e, dentro dos limites, reduzir também a taxa de falta de capacidade/conflito e possivelmente a taxa de falta de compartilhamento verdadeiro — é útil para essa carga de trabalho. A Figura 5.15 mostra o número de faltas por 1.000 instruções à medida que o tamanho do bloco é aumentado de 32 para 256. Aumentar o tamanho do bloco de 32 para 256 afeta quatro dos componentes da taxa de falta:

- A taxa de falta de compartilhamento verdadeiro diminui por um fator de mais de 2, indicando alguma proximidade nos padrões de compartilhamento verdadeiro.
- A taxa de falta compulsória diminui bastante, como poderíamos esperar.
- As faltas de conflito/capacidade mostram uma pequena diminuição (um fator de 1,26 em comparação com um fator de aumento de 8 no tamanho do bloco), indicando que a localidade espacial não é alta nas faltas de uniprocessador que ocorrem com caches L3 maiores que 2 MB.
- A taxa de falta de compartilhamento falso, embora pequena em termos absolutos, quase dobra.

A falta de um efeito significativo na taxa de falta de instrução é surpreendente. Se houvesse uma cache de única instrução com esse comportamento, concluiríamos que a localidade espacial é muito pobre. No caso de uma cache L2 misturada, outros efeitos, como conflitos de instrução-dados, também podem contribuir para a alta taxa de faltas de cache de instrução para blocos maiores. Outros estudos têm documentado a baixa localidade espacial no fluxo de instruções de grandes cargas de trabalho de banco de dados e OLTP, que possuem muitos blocos básicos curtos e sequências de código de uso especial. Com base nesses dados, a penalidade de falta para que uma L3 com tamanho de bloco grande

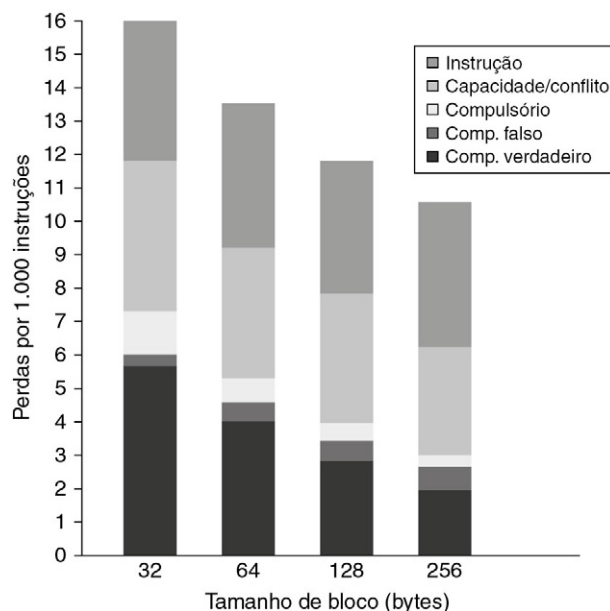



FIGURA 5.15 O número de faltas por 1.000 instruções cai fortemente à medida que o tamanho de bloco da cache L3 aumenta, criando um bom caso para um tamanho de bloco L3 de pelo menos 128 bytes.
A cache L3 é uma cache de 2 MB, associativa por conjunto de duas vias.

tenha desempenho tão bom quanto a L3 com tamanho de bloco de 32 bytes pode ser expressa como um multiplicador sobre a penalidade do bloco de tamanho de 32 bytes:

Tamanho de bloco	Penalidade de falta relativa à penalidade de falta do bloco de 32 bytes
64 bytes	1,19
128 bytes	1,36
256 bytes	1,52

Com SDRAMs DDR modernas, que tornam o acesso aos blocos mais rápido, esses números parecem alcançáveis, especialmente no tamanho de bloco de 128 bytes. Obviamente, devemos também nos preocupar com os efeitos do maior tráfego para a memória e a possível disputa pela memória com outros núcleos. Este último efeito pode negar facilmente os ganhos obtidos com a melhora do desempenho de um único processador.

Uma carga de trabalho de multiprogramação e de SO

Nosso próximo estudo é uma carga de trabalho multiprogramada que consiste em atividades do usuário e atividades do SO. A carga de trabalho usada são duas cópias independentes das fases de compilação do benchmark Andrew, que simula um ambiente de desenvolvimento de software. A fase de compilação consiste em uma versão paralela do "make" do UNIX executada usando oito processadores. A carga de trabalho é executada por 5,24 segundos em oito processadores, criando 203 processos e realizando 787 solicitações de disco em três sistemas de arquivo diferentes. A carga de trabalho é executada com 128 MB de memória, e nenhuma atividade de paginação acontece.

A carga de trabalho possui três fases distintas: 1) compilação dos benchmarks, que envolve atividade substancial de computação; 2) instalação dos arquivos objetos em uma biblioteca; e 3) remoção dos arquivos objetos. A última fase é completamente dominada pela E/S, e somente dois processos estão ativos (um para cada uma das execuções). Na fase do meio, a E/S também desempenha um papel importante, e o processador fica em grande parte ocioso. A carga de trabalho geral é muito mais intensiva para o sistema e para E/S do que a carga de trabalho comercial altamente ajustada.

Para as medições de carga de trabalho, consideramos os seguintes sistemas de memória e E/S:

- *Cache de instrução de nível 1.* 32 KB, associativa por conjunto de duas vias, com um bloco de 64 bytes, tempo de acerto de um ciclo de clock.
- *Cache de dados de nível 1.* 32 KB, associativa por conjunto de duas vias, com um bloco de 32 bytes, tempo de acerto de um ciclo de clock. Variamos a cache de dados L1 para examinar seu efeito sobre o comportamento da cache.
- *Cache de nível 2.* 1 MB unificado, associativa por conjunto de duas vias, com um bloco de 128 bytes, tempo de acerto de 10 ciclos de clock.
- *Memória principal.* Única memória em um barramento com tempo de acesso de 100 ciclos de clock.
- *Sistema de disco.* Latência de acesso fixa de 3 ms (menos do que o normal para reduzir o tempo ocioso).

A Figura 5.16 mostra como o tempo de execução é desmembrado para os oito processadores que usam os parâmetros que listamos. O tempo de execução é desmembrado em quatro componentes:

1. *Ocioso.* Execução no loop ocioso do modo kernel.
2. *Usuário.* Execução no código do usuário.

	Execução do usuário	Execução do kernel	Espera de sincronização	Processador ocioso (esperando E/S)
Instruções executadas	27%	3%	1%	69%
Tempo de execução	27%	7%	2%	64%

FIGURA 5.16 **Distribuição do tempo de execução na carga de trabalho do "make" paralelo multiprogramado.** A alta fração de tempo ocioso é devida à latência do disco quando apenas um dos oito processadores está ativo. Esses dados e as medições subsequentes para essa carga de trabalho foram coletados com o sistema SimOS (Rosenblum *et al.*, 1995). As execuções reais e a coleta de dados foram feitas por M. Rosenblum, S. Herrod e E. Bugnion, da Stanford University.

3. *Sincronização.* Execução ou esperando por variáveis de sincronismo.
4. *Kernel.* Execução no SO que não é ociosa nem no acesso de sincronização.

Essa carga de trabalho de multiprogramação possui significativa perda de desempenho da cache de instrução, pelo menos para o SO. A taxa de falta da cache de instrução no SO para uma cache associativa por conjunto de duas vias, com tamanho de bloco de 64 bytes, varia de 1,7% para uma cache de 32 KB até 0,2% para uma cache de 256 KB. As faltas da cache de instrução em nível de usuário são aproximadamente 1/6 da taxa do SO, para uma série de tamanhos de cache. Isso é devido, parcialmente, ao fato de que, embora o código do usuário execute nove vezes mais instruções do que o kernel, essas instruções levam apenas cerca de quatro vezes o tempo do menor número de instruções executadas pelo kernel.

Desempenho da carga de trabalho da multiprogramação e do SO

Nesta subseção, examinaremos o desempenho de cache da carga de trabalho multiprogramada à medida que o tamanho da cache e o tamanho do bloco são alterados. Devido às diferenças entre o comportamento do kernel e os processos do usuário, mantemos esses dois componentes separados. No entanto, lembre-se de que os processos do usuário executam cerca de oito vezes mais instruções, de modo que a taxa de falta geral é determinada sobretudo pela taxa de falta no código do usuário, que, conforme veremos, normalmente é 1/5 da taxa de falta do kernel.

Embora o código do usuário execute mais instruções, o comportamento do sistema operacional pode causar mais faltas de cache do que os processos do usuário por dois motivos, além do maior tamanho de código e da falta de localidade: 1) o kernel inicializa todas as páginas antes de alocá-las a um usuário, o que aumenta significativamente o componente compulsório da taxa de falta do kernel; 2) na realidade, o kernel compartilha dados e possui taxa de falta coerente não trivial. Ao contrário, os processos do usuário só causam faltas de coerência quando o processo é escalonado em um processador diferente, e esse componente da taxa de falta é pequeno.

A Figura 5.17 mostra a taxa de falta de dados contra o tamanho da cache de dados e contra o tamanho do bloco para os componentes de kernel e usuário. Aumentar o tamanho da cache de dados afeta a taxa de falta do usuário mais do que a taxa de falta do kernel. Aumentar o tamanho do bloco tem efeitos benéficos para as duas taxas de faltas, pois uma fração maior das faltas surge do compulsório e da capacidade, e ambos podem ser melhorados com tamanhos de bloco maiores. Como as faltas de coerência são relativamente mais raras, os efeitos negativos de aumentar o tamanho de bloco são pequenos. Para entender por que os processos do kernel e do usuário se comportam de forma diferente, podemos examinar como se comportam as faltas do kernel.

A Figura 5.18 mostra a variação nas faltas do kernel contra os aumentos no tamanho de cache e no tamanho do bloco. As faltas são desmembradas em três classes: faltas compulsórias, faltas de coerência (de compartilhamento verdadeiro e falso) e faltas de capacidade/

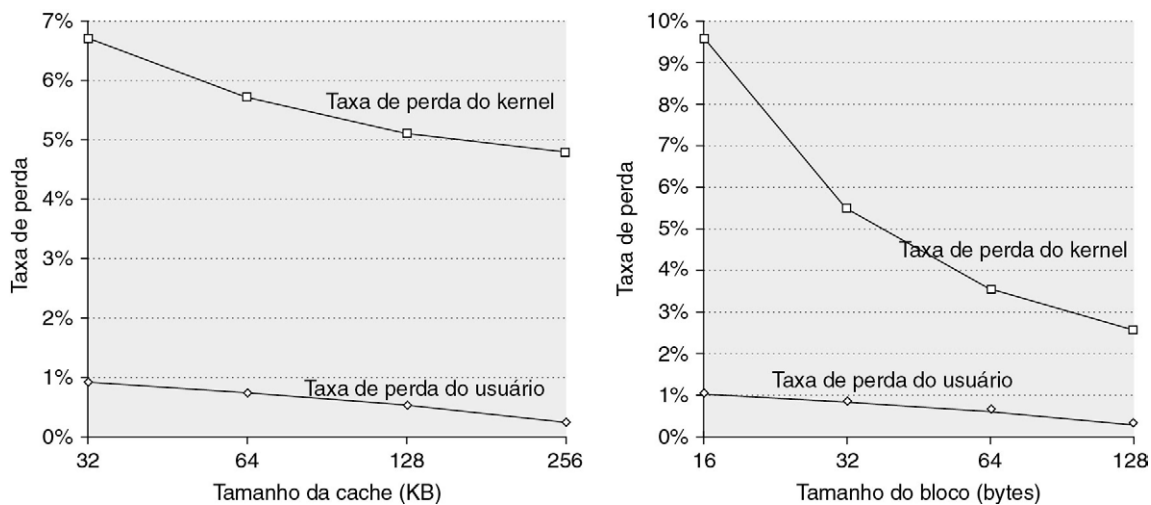



FIGURA 5.17 As taxas de falta de dados para os componentes usuário e kernel se comportam de formas diferentes para aumentos no tamanho da cache de dados L1 (à esquerda) *versus* aumentos no tamanho de bloco da cache de dados L1 (à direita). Aumentar a cache de dados L1 de 32 KB para 256 KB (com um bloco de 32 bytes) faz com que a taxa de falta do usuário diminua proporcionalmente mais do que a taxa de falta do kernel: a taxa de falta em nível de usuário cai por quase um fator de 3, enquanto a taxa de falta em nível de kernel cai apenas por um fator de 1,3. A taxa de falta para os componentes usuário e kernel cai fortemente à medida que o tamanho do bloco L1 aumenta (enquanto mantém a cache L1 em 32 KB). Ao contrário dos efeitos de aumentar o tamanho da cache, aumentar o tamanho do bloco melhora a taxa de falta do kernel mais significativamente (pouco abaixo de um fator de 4 para as referências ao kernel quando passa de blocos de 16 bytes para 128 bytes *versus* pouco menos de um fator de 3 para as referências do usuário).

conflito (que incluem faltas causadas por interferência entre o SO e o processo do usuário e entre múltiplos processos do usuário). A Figura 5.18 confirma que, para as referências de kernel, o aumento no tamanho da cache reduz unicamente a taxa de falta de capacidade/conflito do uniprocessador. Ao contrário, o aumento no tamanho do bloco causa uma redução na taxa de falta compulsória. A ausência de grandes aumentos na taxa de falta de coerência à medida que o tamanho do bloco é aumentado significa que os efeitos do compartilhamento falso são insignificantes, embora essas faltas possam desviar alguns dos ganhos advindos da redução das faltas reais de compartilhamento.

Se examinarmos o número de bytes necessários por referência de dados, como na Figura 5.19, veremos que o kernel possui uma razão de tráfego mais alta, que cresce com o tamanho do bloco. É fácil ver por que isso acontece: ao passar de um bloco de 16 bytes para um bloco de 128 bytes, a taxa de falta cai por volta de 3,7, mas o número de bytes transferidos por falta aumenta em 8, de modo que o tráfego de falta total aumenta por um fator um pouco maior que 2. O programa do usuário também mais que dobra quando o tamanho do bloco vai de 16 para 128 bytes, mas ele começa em um nível muito mais baixo.

Para a carga de trabalho multiprogramada, o SO é um usuário muito mais exigente do sistema de memória. Se mais atividade do SO ou do tipo de SO for incluída na carga de trabalho e o comportamento for semelhante ao que foi medido para essa carga de trabalho, ficará muito difícil construir um sistema de memória suficientemente capaz. Um possível caminho para melhorar o desempenho é tornar o SO mais consistente em termos de cache através de melhores ambientes de programação ou através da assistência do programador. Por exemplo, o SO reutiliza a memória para solicitações que surgem de diferentes chamadas do sistema. Apesar do fato de a memória reutilizada ser completamente sobrescrita, o hardware, não reconhecendo isso, tentará preservar a coerência e a possibilidade de que alguma parte de um bloqueio de cache possa ser lida, mesmo que não seja. Esse comportamento é semelhante à reutilização de locais de pilha nas chamadas

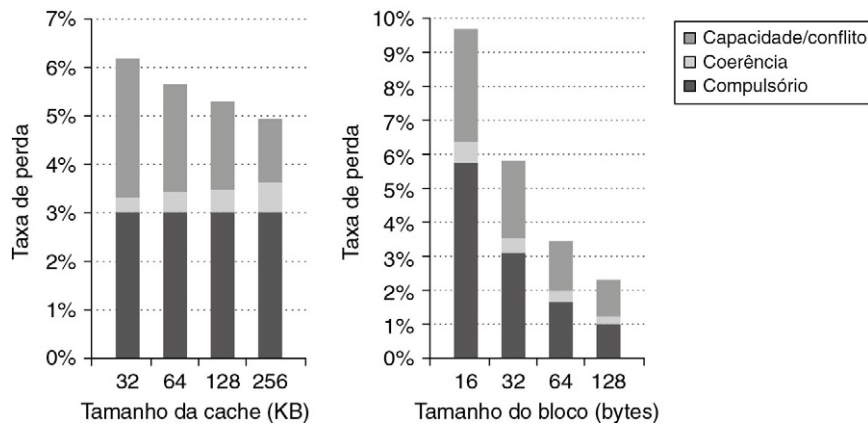



FIGURA 5.18 Os componentes da taxa de falta de dados do kernel mudam à medida que o tamanho da cache de dados L1 aumenta de 32 KB para 256 KB, quando a carga de trabalho de multiprogramação é executada em oito processadores.

O componente de taxa de falta compulsória permanece constante, pois não é afetado pelo tamanho da cache. O componente de capacidade cai por um fator maior que 2, enquanto o componente de coerência quase dobra. O aumento nas faltas de coerência acontece porque a probabilidade de uma falta ser causada por uma invalidação aumenta com o tamanho da cache, visto que menos entradas são colididas devido à capacidade. Como poderíamos esperar, o tamanho maior de bloco da cache de dados L1 reduz substancialmente a taxa de falta compulsória nas referências do kernel. Isso também tem um impacto significativo sobre a taxa de falta de capacidade, reduzindo-a por um fator de 2,4 pelo intervalo de tamanhos de bloco. O tamanho maior de bloco tem uma pequena redução no tráfego de coerência, que parece estabilizar em 64 bytes, sem mudança na taxa de falta de coerência passando para as linhas de 128 bytes. Como não existem reduções significativas na taxa de falta de coerência quando o tamanho de bloco aumenta, a fração da taxa de falta devido à coerência aumenta de cerca de 7% para cerca de 15%.

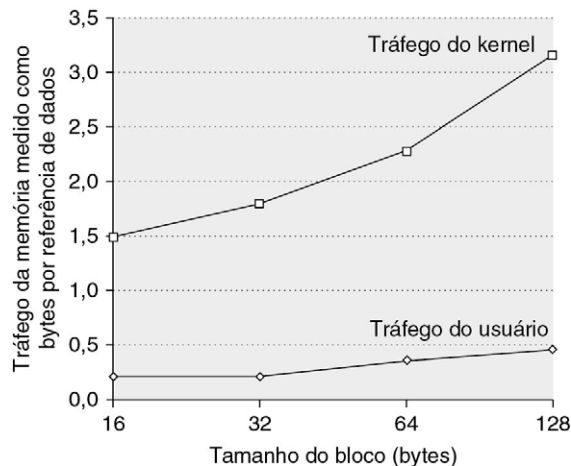



FIGURA 5.19 O número de bytes necessários por referência de dados cresce à medida que o tamanho do bloco aumenta para os componentes do kernel e do usuário.

É interessante comparar esse gráfico com os dados nos programas científicos mostrados no Apêndice I.

de procedimento. A série Power da IBM possui suporte para permitir que o compilador indique esse tipo de comportamento nas chamadas de procedimento, e os processadores AMD mais recentes têm suporte similar. É mais difícil detectar esse comportamento pelo SO, e isso pode exigir assistência do programador, mas a recompensa é potencialmente ainda maior.

As cargas de trabalho comerciais e de SO apresentam desafios difíceis para os sistemas de memória de multiprocessadores e, ao contrário das aplicações científicas, que vamos examinar no Apêndice I, elas são menos adequadas à reestruturação algorítmica ou de compilador. Conforme o número de núcleos aumenta, prever o comportamento de tais aplicações provavelmente vai ficar mais difícil. As metodologias de emulação ou simulação que permitem a simulação de centenas de núcleos com grande variedade de aplicações (incluindo sistemas operacionais) serão cruciais para manter uma abordagem analítica e quantitativa para o projeto.

5.4 MEMÓRIA DISTRIBUÍDA COMPARTILHADA E COERÊNCIA BASEADA EM DIRETÓRIO

Como vimos na Seção 5.2, um protocolo de snooping exige comunicação com todas as caches em cada falta de cache, incluindo as escritas de dados potencialmente compartilhados. A ausência de qualquer estrutura de dados centralizada que acompanhe o estado das caches é tanto uma vantagem fundamental de um esquema baseado em snooping, pois permite que ele seja pouco dispendioso, quanto seu calcanhar de Aquiles quando se trata de escalabilidade.

Considere, por exemplo, um multiprocessador composto de quatro multicores de quatro núcleos capaz de sustentar uma referência de dados por clock e um clock de 4 GHz. Com base nos dados na Seção I.5 do Apêndice I, poderemos ver que as aplicações podem requerer de 4 GB/s a 170 GB/s de largura de banda de barramento. Embora as caches nesses experimentos sejam pequenas, a maior parte do tráfego é coerente e não é afetado pelo tamanho da cache. Embora um barramento moderno possa acomodar 4 GB/s, 170 GB/s vai muito além da capacidade de qualquer sistema baseado em barramento. Nos últimos anos, o desenvolvimento de processadores multicore forçou todos os projetistas a mudarem para alguma forma de memória distribuída para suportar as demandas de largura de banda dos processadores individuais.

Podemos aumentar a largura de banda da memória e a largura de banda entre conexões distribuindo a memória, como mostra a Figura 5.2, na página 305; isso separa imediatamente o tráfego de memória local do tráfego de memória remota, reduzindo as demandas de largura de banda no sistema de memória e na rede de interconexão. A menos que eliminemos a necessidade do protocolo de coerência ser transmitido por broadcast em cada falta de cache, a distribuição da memória nos dará pouco ganho.

Conforme mencionamos, a alternativa a um protocolo de coerência baseado em snooping é um *protocolo de diretório*. Um diretório mantém o estado de cada bloco que pode ser mantido na cache. As informações no diretório incluem as caches (ou grupos de caches) que possuem cópias do bloco, se ele está modificado, e assim por diante. Dentro de um multicore com uma cache externa compartilhada (L3, por exemplo) é fácil implementar um esquema de diretório: simplesmente mantenha um vetor de bits de tamanho igual ao número de núcleos para cada bloco L3. O vetor de bits indica que caches privativas podem ter cópias de um bloco em L3, e as invalidações são enviadas somente para essas caches. Isso funciona perfeitamente para um único multicore se L3 for inclusiva, e esse esquema é usado no Intel i7.

A solução de um único diretório usado em um multicore não é escalável, embora evite o broadcast. O diretório deve ser distribuído, mas a distribuição deve ser feita de modo que o protocolo de coerência saiba onde encontrar a informação de diretório para qualquer bloco de memória na cache. A solução óbvia é distribuir o diretório juntamente com a memória, então essas requisições diferentes de coerência podem ir para diferentes diretórios, assim como diferentes requisições de memória vão para memórias diferentes. Uma característica do diretório distribuído é que o estado de compartilhamento de um bloco

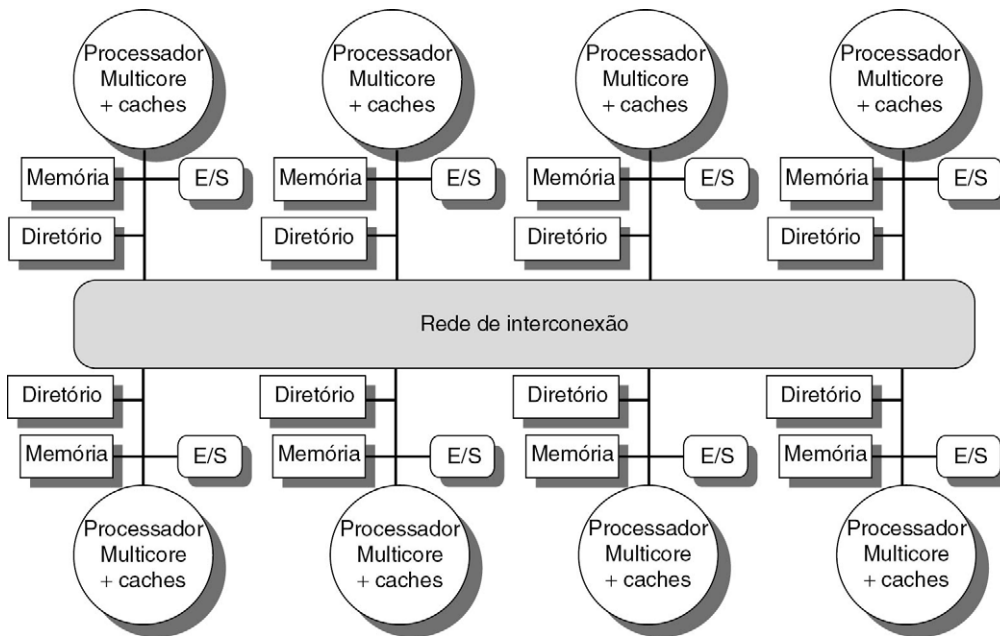



FIGURA 5.20 Um diretório é adicionado a cada nó para implementar a coerência de cache em um multiprocessador de memória distribuída.
Cada diretório é responsável por rastrear as caches que compartilham os endereços de memória da parte da memória no nó. O mecanismo de coerência lidaria com a manutenção do diretório e quaisquer ações de coerência dentro do nó multicore.

está sempre em uma única localização conhecida. Essa propriedade, juntamente com a manutenção da informação que diz que outros nós podem ter o bloco na cache, é o que permite que o protocolo de coerência evite o broadcast. A Figura 5.20 mostra como é o nosso multiprocessador de memória distribuída com os diretórios adicionados a cada nó.

As implementações de diretório mais simples associam uma entrada no diretório a cada bloco de memória. Nessas implementações, a quantidade de informação é proporcional ao produto do número de blocos de memória (em que cada bloco tem o mesmo tamanho do bloco de cache de L2 ou L3) pelo número de processadores. Esse overhead não é um problema para os multiprocessadores com menos de algumas centenas de processadores (cada um dos quais pode ser um multicore), pois o overhead do diretório com tamanho de bloco razoável será tolerável. Para multiprocessadores maiores, precisaremos de métodos para permitir que a estrutura de diretório seja eficientemente escalada, mas somente sistemas do tamanho de supercomputadores precisam se preocupar com isso.

Protocolos de coerência de cache baseados em diretório: fundamentos

Assim como em um protocolo snooping, existem duas operações principais que um protocolo de diretório precisa implementar: tratamento de falta de leitura e tratamento de escrita em um bloco de cache compartilhada, limpa. (O tratamento de falta de escrita em um bloco que está sendo compartilhado é uma simples combinação desses dois.) Para implementar essas operações, um diretório precisa rastrear o estado de cada bloco de cache. Em um protocolo simples, esses estados poderiam ser os seguintes:

- *Compartilhado.* Um ou mais processadores têm o bloco na cache, e o valor na memória está atualizado (assim como em todas as caches).
- *Uncached.* Nenhum processador tem uma cópia do bloco da cache.

- *Modificado*. Um processador tem exatamente uma cópia do bloco da cache, e ele escreveu no bloco, de modo que a cópia na memória está desatualizada. O processador é chamado de *proprietário* (owner) do bloco.

Além de rastrear o estado de cada bloco de memória potencialmente compartilhado, temos que rastrear quais processadores possuem cópias desse bloco, pois essas cópias precisarão ser invalidadas em uma escrita. A maneira mais simples de fazer isso é manter um vetor de bits para cada bloco da memória. Quando o bloco é compartilhado, cada bit do vetor indica se o processador correspondente tem uma cópia desse bloco. Também podemos usar o vetor de bits para acompanhar o proprietário do bloco quando o bloco estiver no estado exclusivo. Por questões de eficiência, também rastreamos o estado de cada bloco de cache nas caches individuais.

Os estados e as transições para a máquina de estado em cada cache são idênticos ao que usamos para a cache de snooping, embora as ações em uma transição sejam ligeiramente diferentes. Os processos de invalidar ou localizar uma cópia exclusiva de um item de dados são diferentes, pois ambos envolvem a comunicação entre o nó solicitante e o diretório, e entre o diretório e um ou mais nós remotos. Em um protocolo snooping, essas duas etapas são combinadas por meio do uso de um broadcast a todos os nós.

Antes de vermos os diagramas de estado de protocolo, é útil examinar um catálogo dos tipos de mensagem que podem ser enviados entre os processadores e os diretórios com a finalidade de tratar as faltas e manter a coerência. A Figura 5.21 mostra o tipo de mensagem enviada entre os nós. O *nó local* é o nó onde uma solicitação se origina. O *nó raiz* é aquele

Tipo de mensagem	Origem	Destino	Conteúdo da mensagem	Função dessa mensagem
Falta de leitura	Cache local	Diretório raiz	*P*, *A*	O nó P tem falta de leitura no endereço *A*; solicitar dados e tornar P um compartilhador de leitura
Falta de escrita	Cache local	Diretório raiz	*P*, *A*	O nó P tem falta de escrita no endereço *A*; solicitar dados e tornar P um proprietário exclusivo.
Invalidação	Cache local	Diretório raiz	*A*	Solicitar para enviar invalidações a todos as caches remotas que estão colocando o bloco na cache no endereço *A*.
Invalidação	Diretório raiz	Cache remoto	*A*	Invalidar uma cópia compartilhada dos dados no endereço *A*.
Busca	Diretório raiz	Cache remoto	*A*	Buscar o bloco no endereço *A* e enviá-lo ao seu diretório raiz; alterar o estado de A na cache remota para compartilhada.
Busca/invalidação	Diretório raiz	Cache remoto	*A*	Buscar o bloco no endereço *A* e enviá-lo ao seu diretório raiz; invalidar o bloco na cache.
Reply de valor de dados	Diretório raiz	Cache local	*D*	Retornar um valor de dados da memória raiz.
Write-back de dados	Cache remoto	Diretório raiz	*A*, *D*	Escrever de volta um valor de dados para o endereço *A*.

FIGURA 5.21 As mensagens possíveis enviadas entre os nós para manter a coerência, junto com o nó de origem e destino, o conteúdo (onde *P* = número do processador solicitante, *A* = endereço solicitado e *D* = conteúdo de dados) e a função da mensagem.
As três primeiras mensagens são solicitações enviadas pelo nó local ao nó raiz. As mensagens quatro a seis são enviadas ao nó remoto pelo nó raiz, quando o nó raiz precisa dos dados para satisfazer uma solicitação de falta de leitura ou escrita. Replies de valor de dados são usadas para enviar um valor do nó raiz de volta ao nó solicitante. Write-backs do valor de dados ocorrem por dois motivos: quando um bloco é substituído em uma cache e precisa ser escrito de volta em sua memória raiz e também na resposta para buscar ou buscar/invalidar mensagens do nó raiz. O write-back do valor de dados sempre que o bloco se torna compartilhado simplifica o número de estados no protocolo, pois qualquer bloco modificado precisa ser exclusivo e qualquer bloco compartilhado está sempre disponível na memória raiz.

em que residem o local da memória e a entrada de diretório de um endereço. O espaço de endereço físico é distribuído estaticamente, de modo que o nó que contém a memória e o diretório para determinado endereço físico é conhecido. Por exemplo, os bits de alta ordem podem oferecer o número do nó, enquanto os bits de baixa ordem oferecem o deslocamento dentro da memória nesse nó. O nó local também pode ser o nó raiz. O diretório precisa ser acessado quando o nó raiz for o nó local, pois as cópias podem existir em um terceiro nó, chamado *nó remoto*.

Nó remoto é aquele que possui uma cópia de um bloco de cache, seja ele exclusivo (em que é a única cópia), seja compartilhado. Um nó remoto pode ser o mesmo que o nó local ou o nó raiz. Nesses casos, o protocolo básico não muda, mas as mensagens entre os processadores podem ser substituídas por mensagens dentro do processador.

Nesta seção, assumiremos um modelo simples de consistência de memória. Para minimizar o tipo de mensagem e a complexidade do protocolo, estamos supondo que as mensagens serão recebidas e trabalhadas na mesma ordem em que são enviadas. Essa suposição pode não ser verdadeira na prática e resultar em complicações adicionais, algumas das quais abordaremos na Seção 5.6, quando falarmos sobre modelos de consistência de memória. Nesta seção, usaremos essa suposição para garantir que as invalidações enviadas por um processador sejam honradas antes de novas mensagens serem transmitidas, assim como presumimos na análise da implementação dos protocolos snooping. Como fizemos no caso do snooping, omitimos alguns detalhes necessários para implementar o protocolo de coerência. Em particular, a serialização de escritas e o conhecimento de que as invalidações para uma escrita foram completadas não são tão simples quanto no mecanismo de snooping baseado em broadcast. Em vez disso, confirmações explícitas são exigidas em resposta a faltas de escrita e solicitações de invalidação. Tratamos dessas questões com mais detalhes no Apêndice I.

Um exemplo de protocolo de diretório

Os estados básicos de um bloco de cache em um protocolo baseado em diretório são exatamente como aqueles em um protocolo snooping, e os estados no diretório também são semelhantes aos que mostramos anteriormente. Assim, podemos começar com diagramas de estado simples que mostram as transições de estado para um bloco de cache individual e depois examinar o diagrama de estado para a entrada de diretório correspondente a cada bloco na memória. Assim como no caso do snooping, esses diagramas de transição de estado não representam todos os detalhes de um protocolo de coerência; porém, o controlador real é altamente dependente de uma série de detalhes do multiprocessor (propriedades de entrega de mensagem, estruturas de buffers, e assim por diante). Nesta seção, apresentaremos os diagramas de estado de protocolo básico. As questões complicadas envolvidas na implementação desses diagramas de transição de estado serão examinadas no Apêndice I.

A Figura 5.22 mostra as ações de protocolo às quais uma cache individual responde. Usaremos a mesma notação da seção anterior, com as solicitações vindas de fora do nó em cinza e as ações em negrito. As transições de estado para uma cache individual são causadas por faltas de leitura, faltas de escrita, invalidações e solicitações de busca de dados; todas essas operações aparecem na Figura 5.22. Uma cache individual também gera mensagens de falta de leitura, falta de escrita e invalidação, que são enviadas ao diretório raiz. Faltas de leitura e escrita exigem respostas de valor de dados, e esses eventos esperam respostas antes de alterar o estado. Saber quando as invalidações são concluídas é um problema isolado, sendo tratado separadamente.

A operação do diagrama de transição de estado para um bloco de cache na Figura 5.22 é essencialmente a mesma que para o caso snooping: os estados são idênticos e o estímulo é

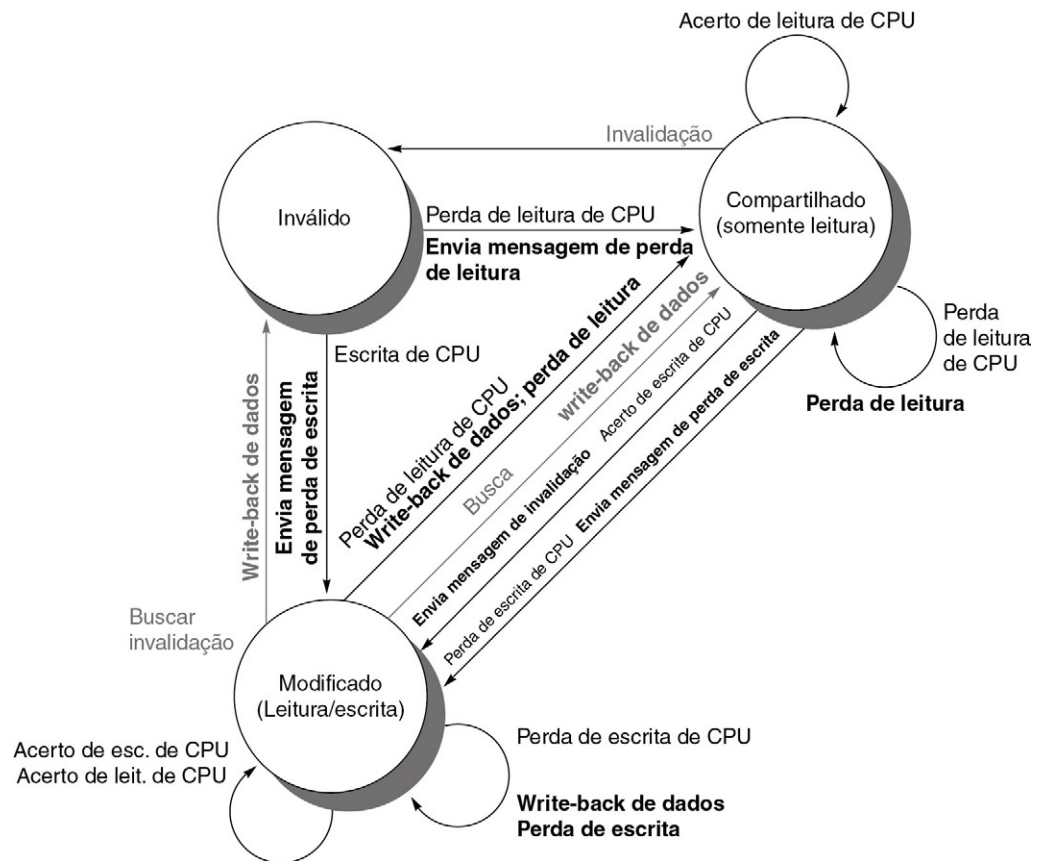



FIGURA 5.22 Diagrama de transição de estado para um bloco de cache individual em um sistema baseado em diretório.

As solicitações do processador local aparecem em preto e as do diretório raiz aparecem em cinza. Os estados são idênticos àqueles do caso snooping, e as transições são muito semelhantes, com solicitações explícitas de invalidação e write-back substituindo as faltas de escrita que foram enviadas por broadcast no barramento. Como fizemos para o controlador de snooping, assumimos que uma tentativa de escrita de um bloco de cache compartilhado é tratada como falta; na prática, essa transação pode ser tratada como uma solicitação de propriedade ou solicitação de atualização, e pode oferecer propriedade sem exigir que o bloco de cache seja buscado.

quase idêntico. A operação de falta de escrita, que foi enviada por broadcast no barramento (ou outra rede) no esquema snooping, é substituída pelas operações de busca de dados e invalidação, que são enviadas seletivamente pelo controlador de diretório. Assim como o protocolo snooping, qualquer bloco de cache precisa estar no estado exclusivo quando for escrito, e qualquer bloco compartilhado precisa estar atualizado na memória. Em muitos processadores multicore, o nível mais externo na cache do processador é compartilhado entre os núcleos (como o L3 no Intel i7, o AMD Opteron e o IBM Power7), e o hardware nesse nível mantém a coerência entre as caches privadas de cada núcleo no mesmo chip, usando um diretório interno ou snooping. Assim, o mecanismo de coerência no chip multicore pode ser usado para estender a coerência entre um conjunto maior de processadores simplesmente fazendo a interface para a cache compartilhada mais externa. Como essa interface está no L3, a contenção entre o processador e as requisições de coerência de um despacho, e a duplicação das tags podem ser evitadas.

Em um protocolo baseado em diretório, o diretório implementa a outra metade do protocolo de coerência. Uma mensagem enviada a um diretório causa dois tipos diferentes

de ações: 1) atualizar o estado do diretório; 2) enviar mensagens adicionais para satisfazer a solicitação. Os estados no diretório representam os três estados-padrão para um bloco; porém, diferentemente de um esquema snoopy, o estado do diretório indica o estado de todas as cópias na cache de um bloco da memória, em vez de um único bloco de cache.

O bloco de memória pode ser retirado da cache por qualquer nó, colocado na cache em múltiplos nós e se tornar passível de leitura (compartilhado) ou colocado na cache exclusivamente e passível de escrita em exatamente um nó. Além do estado de cada bloco, o diretório precisa rastrear o conjunto de processadores que possuem uma cópia de um bloco; usamos um conjunto chamado *Sharers* para realizar essa função. Em multiprocessadores com menos de 64 nós (cada qual podendo representar duas a quatro vezes a quantidade de processadores), esse conjunto normalmente é mantido como um vetor de bits. Em multiprocessadores maiores, outras técnicas são necessárias. As solicitações de diretório precisam atualizar o conjunto Sharers e também ler o conjunto para realizar invalidações.

A Figura 5.23 mostra as ações realizadas no diretório em resposta às mensagens recebidas. O diretório recebe três solicitações diferentes: falta de leitura, falta de escrita e write-back de dados. As mensagens enviadas em resposta pelo diretório aparecem em negrito, enquanto a atualização do conjunto Sharers aparece em negrito e itálico. Como todas as mensagens de estímulo são externas, as ações aparecem em cinza. Nosso protocolo simplificado assume que algumas ações são indivisíveis, como a solicitação de um valor e seu envio para outro nó; uma implementação realista não pode usar essa suposição.

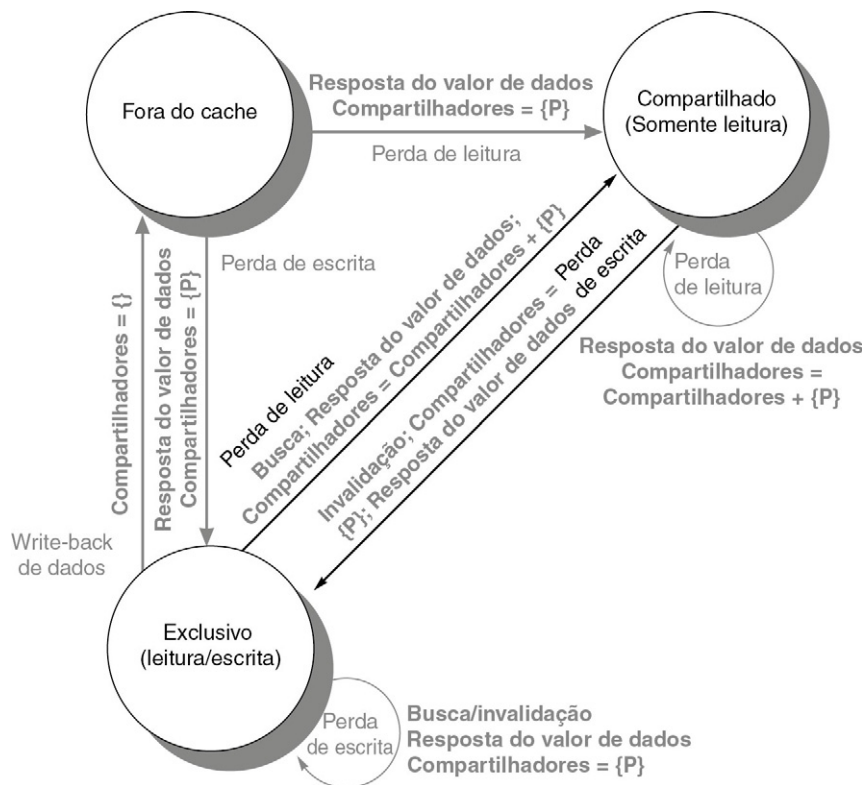



FIGURA 5.23 O diagrama de transição de estado para o diretório tem os mesmos estado e estrutura do diagrama de transição para uma cache individual.

Todas as ações estão em cinza, pois são causadas externamente. O negrito indica a ação tomada pelo diretório em resposta à solicitação.

Para entender essas operações de diretório, vamos examinar as solicitações recebidas e as ações tomadas estado por estado. Quando um bloco está no estado "uncached", a cópia na memória é o valor atual, de modo que as únicas solicitações possíveis para esse bloco são:

- *Falta de leitura.* O processador solicitante recebe os dados solicitados da memória, e o solicitante se torna o único nó de compartilhamento. O estado do bloco se torna compartilhado.
- *Falta de escrita.* O processador solicitante recebe o valor e torna-se o nó de compartilhamento. O bloco se torna exclusivo para indicar que a única cópia válida está na cache. O Sharers indica a identidade do proprietário.

Quando o bloco está no estado compartilhado, o valor da memória é atualizado, de modo que as mesmas duas solicitações podem ocorrer:

- *Falta de leitura.* O processador solicitante recebe os dados solicitados da memória, e o processador solicitante é acrescentado ao conjunto de compartilhamento.
- *Falta de escrita.* O processador solicitante recebe o valor. Todos os processadores no conjunto Sharers recebem mensagens de invalidação, e o conjunto Sharers deve conter a identidade do processador solicitante. O estado do bloco se torna exclusivo.

Quando o bloco está no estado exclusivo, o valor atual do bloco é mantido na cache do processador identificado pelo conjunto Sharers (o proprietário), de modo que existem três solicitações de diretório possíveis:

- *Falta de leitura.* O processador proprietário recebe uma mensagem de busca de dados, que faz com que o estado do bloco na cache do proprietário passe para compartilhado e que o proprietário envie os dados para o diretório, onde são escritos na memória e enviados de volta ao processador solicitante. A identidade do processador solicitante é acrescentada ao conjunto Sharers, que ainda contém a identidade do processador que foi o proprietário (pois ainda tem uma cópia passível de leitura).
- *Write-back de dados.* O processador proprietário está substituindo o bloco e, portanto, precisa escrevê-lo de volta. Essa escrita de volta torna a cópia na memória atualizada (o diretório raiz se torna essencialmente o proprietário), o bloco agora não está na cache e o conjunto Sharers está vazio.
- *Falta de escrita.* O bloco tem um novo proprietário. Uma mensagem é enviada ao proprietário antigo, fazendo com que a cache invalide o bloco e envie o valor ao diretório, do qual é enviado ao processador solicitante, que se torna o novo proprietário. O Sharers é definido como a identidade do novo proprietário, e o estado do bloco permanece exclusivo.

Esse diagrama de transição de estado na Figura 5.23 é uma simplificação, como foi no caso da cache com protocolo snooping. No caso de um diretório, bem como em um esquema snooping implementado com uma rede que não seja um barramento, nossos protocolos precisarão lidar com transações de memória não indivisíveis. O Apêndice I explora essas questões com detalhes.

Os protocolos de diretório usados nos microprocessadores reais contêm otimizações adicionais. Em particular, nesse protocolo, quando ocorre uma falta de leitura ou escrita para um bloco exclusivo, o bloco primeiro é enviado ao diretório no nó raiz. A partir de lá, é armazenado na memória raiz e também enviado ao nó solicitante original. Muitos dos protocolos em uso nos multiprocessadores comerciais encaminham os dados do nó proprietário para o nó solicitante diretamente (além de realizar a escrita de volta para o

nó raiz). Essas otimizações normalmente aumentam a complexidade, aumentando a possibilidade de impasse e os tipos de mensagens que precisam ser tratadas.

A implementação de um esquema de diretório exige a solução da maioria dos mesmos desafios que analisamos para os protocolos snooping, iniciando na página 320. Porém, existem problemas novos e adicionais, que descreveremos no Apêndice I. Na Seção 5.8, descreveremos rapidamente como os multicores modernos estendem a coerência além de um único chip. A combinação de coerência multichip e coerência multicore inclui as quatro possibilidades de snooping/snooping (AMD Opteron), snooping/diretório, diretório/snooping e diretório/diretório!

5.5 SINCRONISMO: FUNDAMENTOS

Os mecanismos de sincronismo normalmente são construídos com rotinas de software em nível de usuário, que contam com instruções de sincronismo fornecidas pelo hardware. Para multiprocessadores menores ou situações de pouca contenção, a principal capacidade do hardware é uma instrução ou sequência de instruções ininterrupta capaz de recuperar e alterar um valor de forma indivisível. Os mecanismos de sincronismo de software são então construídos com a utilização dessa capacidade. Nesta seção, enfocaremos a implementação de operações de sincronismo de bloqueio e desbloqueio. Bloqueio e desbloqueio podem ser usados de forma direta para criar a exclusão mútua, além de implementar mecanismos de sincronismo mais complexos.

Em situações de grande contenção, o sincronismo pode se tornar um gargalo de desempenho, porque a contenção introduz atrasos adicionais e a latência é potencialmente maior em tais multiprocessadores. No Apêndice I, comentaremos como os mecanismos de sincronismo básicos desta seção podem ser estendidos para grande quantidade de processadores.

Primitivas básicas do hardware

A principal capacidade de que precisamos para implementar o sincronismo em um multiprocessador é um conjunto de primitivas de hardware com a capacidade de ler e modificar um local da memória de forma indivisível. Sem tal capacidade, o custo da montagem das primitivas básicas de sincronismo será muito alta e aumentará à medida que o número de processadores aumentar. Existem diversas formulações alternativas das primitivas básicas de hardware, todas oferecendo a capacidade de ler e modificar um local de forma indivisível, junto com alguma maneira de saber se a leitura e a escrita foram realizadas de forma indivisível. Essas primitivas de hardware são os blocos básicos de montagem usados para criar uma variedade de operações de sincronismo em nível de usuário, incluindo itens como bloqueios e barreiras. Em geral, os arquitetos não esperam que os usuários empreguem as primitivas básicas de hardware, mas esperam que as primitivas sejam usadas por programadores de sistemas para criar uma biblioteca de sincronismo, um processo que normalmente é complexo e intrincado. Vamos começar com uma primitiva de hardware desse tipo e mostrar como ela pode ser usada para montar algumas operações básicas de sincronismo.

Uma operação típica para a montagem de operações de sincronismo é a *troca atômica*, que permuta um valor em um registrador por um valor na memória. Para ver como usar isso para montar uma operação básica de sincronismo, suponha que queiramos montar um bloqueio simples, em que o valor 0 seja usado para indicar que o bloqueio está livre e 1 seja usado para indicar que o bloqueio não está disponível. Um processador tenta definir o bloqueio mediante uma troca do 1, que está em um registrador, pelo endereço de memória correspondente ao bloqueio. O valor retornado da instrução de troca é 1 se

algum outro processador já reivindicou acesso e 0 em caso contrário. Neste último caso, o valor também é trocado para 1, impedindo que qualquer troca concorrente apanhe um 0.

Por exemplo, considere dois processadores que tentam realizar a troca simultaneamente: essa corrida será violada, pois exatamente um dos processadores realizará a troca primeiro, retornando 0, e o segundo processador retornará 1 quando fizer a troca. A chave para usar a primitiva de troca (ou swap) para implementar o sincronismo é que a operação seja indivisível: a troca é indivisível, e duas trocas simultâneas serão ordenadas pelos mecanismos de serialização de escrita. É impossível que dois processadores que tentam definir a variável de sincronismo dessa maneira pensem que os dois definiram a variável simultaneamente.

Existem diversas outras primitivas indivisíveis que podem ser usadas para implementar o sincronismo. Todas elas têm a propriedade-chave de que leem e atualizam um valor da memória de tal maneira que podemos saber se as duas operações são ou não executadas indivisivelmente. Uma operação presente em muitos multiprocessadores mais antigos é *testar e marcar*, que testa um valor e o marca se ele passar no teste. Por exemplo, poderíamos definir uma operação que testasse 0 e marcasse o valor como 1, que pode ser usado em um padrão semelhante àquele em que usamos a troca indivisível. Outra primitiva de sincronismo indivisível é *buscar e incrementar*: ela retorna o valor de um local da memória e o incrementa indivisivelmente. Usando o valor 0 para indicar que a variável de sincronismo não é reivindicada, podemos usar buscar e incrementar, assim como usamos a troca. Existem outros usos de operações como buscar e incrementar, que veremos em breve.

A implementação de uma única operação de memória indivisível introduz alguns desafios, pois exige leitura de memória e escrita em uma única instrução que não seja interrompida. Esse requisito complica a implementação da coerência, pois o hardware não pode permitir quaisquer outras operações entre a leitura e a escrita, e também não pode causar impasse.

Uma alternativa é ter um par de instruções na qual a segunda instrução retorne um valor do qual ele possa ser deduzido se o par de instruções for executado como se elas fossem indivisíveis. O par de instruções é efetivamente indivisível se todas as outras operações executadas por qualquer processador parecerem ocorrer antes ou depois do par. Assim, quando um par de instruções é efetivamente indivisível, nenhum outro processador pode alterar o valor no par de instruções.

O par de instruções inclui um carregamento especial, chamado *carregamento ligado* ou *carregamento bloqueado*, e um armazenamento especial chamado *armazenamento condicional*. Essas instruções são usadas em sequência: se o conteúdo do local de memória especificado por um carregamento ligado for alterado antes que ocorra o armazenamento condicional no mesmo endereço, o armazenamento condicional falhará. Se o processador realizar uma troca de contexto entre as duas instruções, o armazenamento condicional também falhará. O armazenamento condicional é definido para retornar 1 se teve sucesso e 0 em caso contrário. Como o carregamento ligado retorna o valor inicial e o armazenamento condicional retorna 1 somente se tiver sucesso, a sequência a seguir implementa uma troca indivisível no local de memória especificado pelo conteúdo de R1:

```
try:  MOV R3,R4   ;move valor de troca
      LL  R2,0(R1);carregamento ligado
      SC  R3,0(R1);armazenamento condicional
      BEQZ R3,try ;desvio se armazenamento falha
      MOV R4,R2   ;coloca valor carregado em R4
```

Ao final dessa sequência, o conteúdo de R4 e o local de memória especificado por R1 foram trocados indivisivelmente (ignorando qualquer efeito dos delayed branches). Sempre que um processador intervém e modifica o valor na memória entre as instruções LL e SC, o SC retorna 0 em R3, fazendo com que a sequência de código tente novamente.

Uma vantagem do mecanismo de carregamento linked/store condicional é que ele pode ser usado para montar outras primitivas de sincronismo. Por exemplo, aqui está um buscar e incrementar indivisível:

```
try:  LL   R2,0(R1) ;carregamento ligado
      DADDUIR3,R2,#1 ;incrementa
      SC   R3,0(R1) ;armazenamento condicional
      BEQZ R3,try  ;desvio se armazenamento falha
```

Essas instruções normalmente são implementadas acompanhando-se o endereço especificado na instrução LL em um registrador, normalmente chamado de *registrador de link*. Se ocorrer uma interrupção ou se o bloco de cache combinado com o endereço no registrador de link for invalidado (p. ex., por outro SC), o registrador de link é apagado. A instrução SC simplesmente verifica se esse endereço combina com o que está no registrador de link. Se combinar, o SC tem sucesso; caso contrário, ele falha. Como o armazenamento condicional falhará após outro armazenamento tentativa para o armazenamento do endereço do carregamento ligado ou qualquer exceção, deve-se ter cuidado na escolha de quais instruções são inseridas entre as duas instruções. Em particular, somente instruções registrador-registrador podem ser permitidas com segurança; caso contrário, é possível criar situações de deadlock nas quais o processador nunca pode completar o SC. Além disso, o número de instruções entre o carregamento ligado e o armazenamento condicional deve ser pequeno para minimizar a probabilidade de que um evento não relacionado ou um processador concorrente cause a falha frequente do armazenamento condicional.

Implementando bloqueios usando a coerência

Em uma operação indivisível, podemos usar os mecanismos de coerência de um multiprocessador para implementar *spin locks* — bloqueios que um processador tenta adquirir continuamente, executando um loop até consegui-lo. Spin locks são usados quando os programadores esperam que o bloqueio seja mantido por um período de tempo muito curto e quando querem que o processo de bloqueio seja de pouca latência quando o bloqueio estiver disponível. Como os spin locks ocupam o processador, esperando em um loop até que o bloqueio seja liberado, são inapropriados em algumas circunstâncias.

A implementação mais simples, que usaríamos se não houvesse coerência de cache, manteria as variáveis de bloqueio na memória. Um processador poderia tentar continuamente adquirir o bloqueio, usando uma operação indivisível, como a troca da página 339, e testar se a troca retornou o bloqueio como livre. Para liberar o bloqueio, o processador simplesmente armazena o valor 0 no bloqueio. Aqui está a sequência de código para bloquear um spin lock cujo endereço está em R1 usando uma troca indivisível:

```
         DADDUIR2,R0,#1
lockit:  EXCHR2,0(R1)    ;troca indivisível
         BNEZR2,lockit   ;já bloqueado?
```

Se o nosso multiprocessador admitir coerência de cache, poderemos colocar os bloqueios na cache usando o mecanismo de coerência para manter o valor do bloqueio coerente-

mente. A colocação dos bloqueios na cache tem duas vantagens. Primeiro, isso permite uma implementação em que o processo de "spinning" (tentar testar e adquirir o bloqueio em um loop curto) poderia ser feito em uma cópia local na cache, em vez de exibir um acesso à memória global em cada tentativa de adquirir o bloqueio. A segunda vantagem vem da observação de que normalmente existe localidade nos acessos de bloqueio, ou seja, o processador que usou o bloqueio por último o usará novamente no futuro próximo. Nesses casos, o valor do bloqueio pode residir na cache desse processador, reduzindo bastante o tempo para adquirir o bloqueio.

Obter a primeira vantagem — ser capaz de girar (spin) em uma cópia local na cache em vez de gerar uma solicitação de memória para cada tentativa de adquirir o bloqueio — exige uma mudança em nosso procedimento simples de spin. Cada tentativa de troca no loop diretamente acima exige uma operação de escrita. Se múltiplos processadores tentarem apanhar o bloqueio, cada qual gerará a escrita. A maioria dessas escritas levará a faltas de escrita, pois cada processador está tentando obter a variável de bloqueio em um estado exclusivo.

Assim, deveremos modificar nosso procedimento de spin lock de modo que ele gire, realizando leituras em uma cópia local do bloqueio, até que veja com sucesso que o bloqueio está disponível. Depois, ele tenta ler a variável de bloqueio que realiza uma operação de swap. Um processador primeiro lê a variável de bloqueio para testar seu estado. Um processador continua lendo e testando até que o valor da leitura indique que o bloqueio está desbloqueado. O processador então corre contra todos os outros processos que estavam "esperando em spin" de modo semelhante, para ver quem pode bloquear primeiro a variável. Todos os processos usam uma instrução de swap que lê o valor antigo e armazena um 1 na variável de bloqueio. O único vencedor verá o 0, e os perdedores verão um 1 que foi colocado lá pelo vencedor (os perdedores continuarão a definir a variável com o valor bloqueado, mas isso não importa). O processador que vence executa o código após o bloqueio e, quando termina, armazena um 0 na variável de bloqueio para liberá-lo, o que inicia a corrida novamente. Aqui está o código para realizar esse spin lock (lembre-se de que 0 é desbloqueado e 1 é bloqueado):

```
lockit:  LDR2,0(R1)      ;carregamento do bloqueio
         BNEZR2,lockit   ;não disponível - spin
         DADDUIR2,R0,#1  ;carrega valor bloqueado
         EXCHR2,0(R1)    ;swap
         BNEZR2,lockit   ;desvia se bloqueio não foi 0
```

Vamos examinar como esse esquema de "spin lock" usa os mecanismos de coerência de cache. A Figura 5.24 mostra as operações de processador e barramento ou diretório para múltiplos processadores tentando bloquear uma variável usando uma troca indivisível. Uma vez que o processador com o bloqueio armazene um 0 no bloqueio, todas as outras caches são invalidadas e devem buscar o novo valor para atualizar sua cópia do bloqueio. Uma dessas caches obtém a cópia do valor não bloqueado (0) primeiro e realiza a troca. Quando a falta na cache de outros processadores é atendida, eles descobrem que a variável já está bloqueada, então eles devem retornar para os testes e para o "spinning".

Esse bloqueio mostra outra vantagem de armazenamento das primitivas condicionais linked/store: as operações de leitura e escrita são separadas explicitamente. O carregamento vinculado não precisa gerar nenhum tráfego de barramento. Esse fato permite a sequência

Etapa	Processador P0	Processador P1	Processador P2	Estado de coerência do bloqueio ao final da etapa	Atividade do barramento/ diretório
1	Tem bloqueio	Spin, testando se bloqueio = 0	Spins, testando se bloqueio = 0	Compartilhado	Nenhum
2	Define bloqueio como 0	(Invalidação recebida)	(Invalidação recebida)	Exclusivo (P0)	Invalidação de escrita da variável de bloqueio de P0
3		Falta na cache	Falta na cache	Compartilhado	Barramento/diretório atende falta na cache de P2; write-back de P0
4		(Espera enquanto barramento/diretório ocupado)	Bloqueio = 0	Compartilhado	Falta na cache para P2 satisfeita
5		Bloqueio = 0	Executa troca, pega falta de cache	Compartilhado	Falta na cache para P1 satisfeita
6		Executa swap, apanha falta na cache	Completa swap: retorna 0 e define bloqueio = 1	Exclusivo (P2)	Barramento/diretório atende falta na cache de P2; gera invalidação
7		Troca completada e retorna 1, e define bloqueio = 1	Entra na seção crítica	Exclusivo (P1)	Barramento/diretório atende falta na cache de P1; gera write-back
8		Spin, testando se bloqueio = 0			Nenhuma

FIGURA 5.24 **Etapas de coerência de cache e tráfego de barramento para três processadores, P0, P1 e P2.**
Essa figura considera a coerência de invalidação de escrita. P0 começa com o bloqueio (etapa 1). P0 sai e retira o bloqueio (etapa 2). P1 e P2 correm para ver qual lê o valor desbloqueado durante o swap (etapas 3 a 5). P2 vence e entra na seção crítica (etapas 6 e 7), enquanto a tentativa de P1 falha, de modo que começa a esperar no spin (etapas 7 e 8). Em um sistema real, esses eventos levarão muito mais do que 8 ticks de clock, pois a aquisição do barramento e a resposta a faltas levam muito mais tempo. Quando a etapa 8 é atingida, o processo pode ser repetido com P2, eventualmente obtendo acesso exclusivo e configurando o bloqueio para 0.

de código simples a seguir, que tem as mesmas características que a versão otimizada usando troca (R1 tem o endereço do bloqueio, o LD foi substituído por LL e SC substituiu EXCH):

```
lockit:  LLR2,0(R1)      ;carregamento linked
         BNEZR2,lockit   ;não disponível-spin
         DADDUIR2,R0,#1  ;valor bloqueado
         SCR2,0(R1)      ;armazena
         BEQZR2,lockit   ;desvia se o armazenamento falhar
```

O primeiro desvio forma o loop spinning. O segundo resolve as corridas quando dois processadores veem o bloqueio disponível ao mesmo tempo.

5.6 MODELOS DE CONSISTÊNCIA DE MEMÓRIA: UMA INTRODUÇÃO

A coerência de cache garante que múltiplos processadores tenham uma visão coerente da memória. Isso não responde à pergunta de *quão* coerente a visão da memória precisa ser. "Quão coerente" significa "quando um processador precisa ver um valor

que foi atualizado por outro processador?". Como os processadores se comunicam por meio de variáveis compartilhadas (usadas tanto para valores de dados quanto para sincronismo), a questão se resume a isto: em que ordem um processador precisa observar as escritas de dados de outro processador? Como a única maneira de observar as escritas de outro processador é através de leituras, a questão se torna: "Que propriedades precisam ser impostas entre leituras e escritas em diferentes locais por diferentes processadores?"

Embora a questão de quão consistente a memória precisa parecer seja simples, ela é notadamente complicada, como podemos ver com um exemplo elementar. Aqui estão dois segmentos de código dos processos P1 e P2, mostrados lado a lado:

```
P1:    A = 0;              P2:    B = 0;
        .....                      .....
       A = 1;                     B = 1;
L1:    if (B == 0)...      L2:    if (A == 0)...
```

Considere que os processos rodam em diferentes processadores e que os locais *A* e *B* são originalmente colocados na cache por ambos os processadores com o valor inicial 0. Se as escritas sempre têm efeito imediato e são vistas imediatamente pelos outros processadores, será impossível para *ambas* as instruções if (rotuladas como L1 e L2) avaliarem suas condições como verdadeiras, pois atingir a instrução if significa que ou *A* ou *B* precisa ter recebido o valor 1. Mas suponha que a invalidação da escrita seja adiada e o processador tenha permissão para continuar durante esse adiamento; então, é possível que nem P1 nem P2 tenham visto as invalidações para *B* e *A* (respectivamente) *antes* de tentar ler os valores. A questão é: esse comportamento deve ser permitido e, se for, sob que condições?

O modelo mais direto de consistência de memória é chamado *consistência sequencial*. A consistência sequencial exige que o resultado de qualquer execução seja como se os acessos à memória executados por processador fossem mantidos em ordem e os acessos entre diferentes processadores fossem intercalados arbitrariamente. A consistência sequencial elimina a possibilidade de alguma execução não óbvia no exemplo anterior, pois as atribuições precisam ser concluídas antes que as instruções if sejam iniciadas.

O modo mais simples de implementar a consistência sequencial é exigir que um processador adie o término de qualquer acesso à memória até que todas as invalidações causadas por esse acesso sejam concluídas. Naturalmente, é igualmente eficaz adiar o próximo acesso à memória até que o anterior seja concluído. Lembre-se de que a consistência da memória envolve operações entre diferentes variáveis: os dois acessos que precisam ser ordenados, na realidade, destinam-se a diferentes locais da memória. Em nosso exemplo, temos que adiar a leitura de *A* ou *B* (A == 0 ou B == 0) até que a escrita anterior tenha sido concluída (B = 1 ou A = 1). Sob a consistência sequencial, não podemos, por exemplo, simplesmente colocar a escrita em um buffer de escrita e continuar a leitura.

Embora a consistência sequencial apresente um paradigma de programação simples, ela reduz o desempenho em potencial, especialmente em um multiprocessador com grande quantidade de processadores ou em grandes atrasos de interconexão, como poderemos ver no exemplo a seguir.

Exemplo Suponha que tenhamos um processador no qual uma falta de escrita leve 50 ciclos para estabelecer a propriedade, 10 ciclos para emitir cada invalidação após a propriedade ser estabelecida e 80 ciclos para uma invalidação concluir e ser confirmada depois de emitida. Supondo que quatro outros processadores compartilhem um bloco de cache, em quanto tempo uma falta de escrita deixará em stall o processador que está escrevendo se o processador for consistente sequencialmente? Considere que as invalidações precisam ser confirmadas explicitamente antes que o controlador de coerência saiba que elas foram concluídas. Suponha que poderíamos continuar executando depois de obter a propriedade para a falta de escrita sem esperar pelas invalidações; quanto tempo a escrita levaria?

Resposta Quando esperamos as invalidações, cada escrita leva a soma do tempo de propriedade mais o tempo para completá-las. Como as invalidações podem se sobrepor, só precisamos nos preocupar com a última, que inicia 10 + 10 + 10 + 10 = 40 ciclos após a propriedade ser estabelecida. Logo, o tempo total para a escrita é 50 + 40 + 80 = 170 ciclos. Em comparação, o tempo de propriedade é de apenas 50 ciclos. Com implementações apropriadas do buffer de escrita, é possível continuar antes que a propriedade seja estabelecida.

Para oferecer melhor desempenho, pesquisadores e arquitetos têm explorado dois caminhos diferentes: 1) eles desenvolveram implementações ambiciosas, que preservam a consistência sequencial, mas utilizam técnicas de ocultação de latência para reduzir a penalidade; discutiremos essas técnicas na Seção 5.7; 2) eles desenvolveram modelos de consistência de memória menos restritivos, que permitem um hardware mais rápido. Esses modelos podem afetar a forma como o programador vê o multiprocessador, de modo que, antes de examinarmos esses modelos menos restritivos, vejamos o que esse programador espera.

A visão do programador

Embora o modelo de consistência sequencial tenha uma desvantagem quanto ao desempenho, do ponto de vista do programador ele tem a vantagem da simplicidade. O desafio é desenvolver um modelo de programação que seja simples de explicar e ainda permita uma implementação de alto desempenho.

Um modelo de programação assim, que nos permita ter uma implementação mais eficiente, deve assumir que os programas estejam *sincronizados*. Um programa está sincronizado quando todos os acessos a dados compartilhados são ordenados por operações de sincronismo. Uma referência de dados é ordenada por uma operação de sincronismo se, em cada execução possível, uma escrita de uma variável por um processador e um acesso (ou uma leitura ou uma escrita) dessa variável por outro processador forem separados por um par de operações de sincronismo, uma executada após a escrita pelo processador que está escrevendo e a outra antes do acesso pelo segundo processador. Casos em que as variáveis podem ser atualizadas sem ordenação pelo sincronismo são chamados *corridas de dados*, pois o resultado da execução depende da velocidade relativa dos processadores e, assim como as corridas no projeto do hardware, o resultado é imprevisível, o que leva a outro nome para os programas sincronizados: *livre de corrida de dados*.

Como exemplo simples, considere uma variável sendo lida e atualizada por dois processadores diferentes. Cada processador cerca a leitura e a atualização com um bloqueio e um desbloqueio, ambos para garantir a exclusão mútua para a atualização e garantir que a leitura seja consistente. Obviamente, agora cada escrita é separada de uma leitura por outro processador por um par de operações de sincronismo: um desbloqueio (após a

escrita) e um bloqueio (antes da leitura). Naturalmente, se dois processadores estiverem escrevendo uma variável sem leituras no intervalo, então as escritas também precisarão ser separadas por operações de sincronismo.

A observação de que a maioria dos programas é sincronizada é bastante aceita. Ela é verdadeira em especial porque, se os acessos não fossem sincronizados, provavelmente o comportamento do programa seria imprevisível, pois a velocidade de execução determinaria qual processador venceu uma corrida de dados e, assim, afetaria os resultados do programa. Mesmo com consistência sequencial, raciocinar sobre tais programas é muito difícil.

Os programadores poderiam tentar garantir a ordenação, construindo seus próprios mecanismos de sincronismo, mas isso é extremamente intrincado, pode levar a programas com erros e não ser aceito arquitetonicamente, significando que eles podem não funcionar em gerações futuras do multiprocessador. Em vez disso, quase todos os programadores decidirão usar bibliotecas de sincronismo corretas e otimizadas para o multiprocessador e o tipo de sincronismo.

Finalmente, o uso de primitivas-padrão de sincronismo garante que, mesmo que a arquitetura implemente um modelo mais relaxado de consistência do que a consistência sequencial, um programa sincronizado se comportará como se o hardware implementasse a consistência sequencial.

Modelos relaxados de consistência: fundamentos

A ideia principal, nos modelos relaxados de consistência, é permitir que leituras e escritas sejam concluídas fora de ordem, mas usando operações de sincronismo para impor a ordenação, de modo que um programa sincronizado se comporte como se o processador fosse sequencialmente consistente. Existem diversos modelos relaxados que são classificados de acordo com as ordenações de leitura e de escrita que eles relaxam. Especificamos as ordenações por um conjunto de regras na forma X→Y, significando que a operação *X* precisa ser concluída antes que a operação *Y* seja realizada. A consistência sequencial exige manter todas as quatro ordenações possíveis: R→W, R→R, W→R e W→W. Os modelos relaxados são definidos pelos quatro conjuntos de ordenações que afrouxam:

1. Relaxar a ordenação W→R gera um modelo conhecido como *ordenação total de armazenamento* ou *consistência de processador*. Como essa ordenação retém a ordenação entre as escritas, muitos programas que operam sob a consistência sequencial operam sob esse modelo, sem sincronismo adicional.
2. Relaxar a ordenação W→W gera um modelo conhecido como *ordenação parcial de armazenamento*.
3. Relaxar as ordenações R→W e R→R gera uma série de modelos, incluindo a *ordenação fraca*, o modelo de consistência do PowerPC, e a *consistência de liberação* (release consistency), dependendo dos detalhes das restrições de ordenação e de como as operações de sincronismo impõem a ordenação.

Relaxando essas ordenações, o processador pode obter vantagens de desempenho significativas. Porém, existem muitas complexidades na descrição dos modelos de consistência relaxados, incluindo as vantagens e as complexidades de relaxar diferentes ordens, definir exatamente o que significa uma conclusão de escrita e decidir quando os processadores podem ver os valores que o próprio processador escreveu. Para obter mais informações sobre as complexidades, questões de implementação e potencial de desempenho dos modelos relaxados, recomendamos fortemente o excelente tutorial de Adve e Gharachorloo (1996).

Comentários finais sobre modelos de consistência

No momento atual, muitos multiprocessadores em construção oferecem suporte para algum tipo de modelo relaxado de consistência, variando da consistência do processador à consistência de liberação. Como o sincronismo é altamente específico do multiprocessador e passível de erro, a expectativa é que a maioria dos programadores usará as bibliotecas de sincronismo-padrão e escreverá programas sincronizados, tornando a escolha de um modelo de consistência fraca invisível ao programador e gerando desempenho mais alto.

Um ponto de vista alternativo, que discutiremos mais extensivamente na próxima seção, argumenta que, com a especulação, grande parte da vantagem do desempenho dos modelos relaxados de consistência pode ser obtida com consistência sequencial ou do processador.

Uma parte importante desse argumento em favor da consistência relaxada gira em torno do papel do compilador e de sua capacidade de otimizar o acesso a variáveis potencialmente compartilhadas na memória; esse assunto também é tratado na Seção 5.7.

5.7 QUESTÕES CRUZADAS

Como os multiprocessadores redefinem muitas características do sistema (p. ex., a avaliação de desempenho, a latência da memória e a importância da escalabilidade), eles introduzem problemas de projeto interessantes, que atravessam o espectro, afetando tanto o hardware quanto o software. Nesta seção, daremos vários exemplos relacionados à questão de consistência da memória. Vamos então examinar o desempenho obtido quando se adiciona multithreading a multiprocessamento.

Otimização do compilador e o modelo de consistência

Outro motivo para definir um modelo para a consistência de memória é especificar o intervalo de otimizações de compilador legais que podem ser realizadas sobre os dados compartilhados. Em programas explicitamente paralelos, a menos que os pontos de sincronismo sejam claramente definidos e os programas sejam sincronizados, o compilador não poderá permutar uma leitura e uma escrita de dois itens de dados compartilhados diferentes, pois essas transformações poderão afetar a semântica do programa. Isso impede até mesmo otimizações relativamente simples, como alocação de registrador de dados compartilhados, pois tal processo normalmente permuta leituras e escritas. Em programas implicitamente paralelos — por exemplo, aqueles escritos em High Performance FORTRAN (HPF) —, os programas precisam ser sincronizados e os pontos de sincronismo são conhecidos, de modo que esse problema não aparece. Se os compiladores podem obter vantagem significativa de modelos de consistência mais relaxados é uma questão aberta, tanto de um ponto de vista de pesquisa, quanto de um ponto de vista prático, em que a falta de modelos uniformes provavelmente vai retardar o progresso de implementação de compiladores.

Usando especulação para esconder a latência nos modelos estritos de consistência

Conforme vimos no Capítulo 3, a especulação pode ser usada para ocultar a latência da memória. Ela também pode ser usada para ocultar a latência que surge de um modelo estrito de consistência, fornecendo muitos dos benefícios de um modelo de memória afrouxado. A principal ideia é que o processador use o escalonamento dinâmico para reordenar as referências de memória, permitindo que elas possam ser executadas fora de ordem. A execução das referências de memória fora de ordem pode gerar violações de consistência sequencial, que poderiam afetar a execução do programa. Essa possibilidade é

evitada com o recurso de confirmação atrasada de um processador especulativo. Considere que o protocolo de consistência é baseado na invalidação. Se o processador receber uma invalidação para uma referência de memória antes que a referência de memória seja confirmada, o processador usará a recuperação de especulação para recuar a computação e reiniciar com a referência de memória cujo endereço foi invalidado.

Se a reordenação das solicitações de memória pelo processador gerar uma ordem de execução que pode levar a um resultado diferente do que teria sido visto sob a consistência sequencial, o processador refará a execução. A chave para o uso dessa técnica é que o processador só precisa garantir que o resultado seja igual como se todos os acessos fossem completados em ordem, e ele pode conseguir isso ao detectar quando os resultados poderiam diferir. A técnica é atraente porque o reinício especulativo raramente será disparado. Ele só será disparado quando houver acessos não sincronizados que realmente causem uma corrida (Gharachorloo, Gupta e Hennessy, 1992).

Hill (1998) defende a combinação de consistência sequencial ou de processador junto com a execução especulativa como modelo de consistência preferido. Seu argumento tem três partes: 1) uma implementação agressiva da consistência sequencial ou de processador ganhará mais com a vantagem de um modelo mais relaxado; 2) tal implementação aumenta muito pouco o custo de implementação de um processador especulativo; 3) essa técnica permite que o programador raciocine usando os modelos de programação mais simples da consistência sequencial ou de processador. A equipe de projeto do MIPS R10000 tinha essa ideia em meados da década de 1990 e usou a capacidade fora de ordem do R10000 para dar suporte a esse tipo de implementação agressiva da consistência sequencial.

Uma questão aberta é o sucesso que a tecnologia de compilador terá na otimização de referências de memória a variáveis compartilhadas. O estado da tecnologia de otimização e o fato de que os dados compartilhados são acessados com frequência por meio de ponteiros ou indexação de array limitaram o uso dessas otimizações. Se essa tecnologia estivesse disponível e levasse a vantagens de desempenho significativas, os escritores de compilador desejariam ser capazes de tirar proveito de um modelo de programação mais relaxado.

Inclusão e sua implementação

Todos os multiprocessadores utilizam hierarquias de cache multinível para reduzir a demanda na interconexão global e a latência de perdas de cache. Se a cache também oferecer *inclusão multinível* — cada nível de hierarquia de cache é um subconjunto do nível mais distante do processador —, então poderemos usar a estrutura multinível para reduzir a disputa entre o tráfego de coerência e o tráfego de processador que ocorre quando os snoops e os acessos à cache do processador precisam disputar a cache. Muitos multiprocessadores com cache multinível impõem a propriedade de inclusão, embora muitas vezes os multiprocessadores recentes, com caches L1 menores e tamanhos de bloco diferentes, escolham não impor a inclusão. Essa restrição também é chamada *propriedade de subconjunto*, pois cada cache é um subconjunto da cache abaixo dela na hierarquia.

À primeira vista, preservar a propriedade de inclusão multinível parece algo trivial. Considere um exemplo em dois níveis: qualquer falta no L1 causa um acerto em L2 ou gera uma falta em L2, fazendo com que seja trazido para L1 e L2. De modo semelhante, qualquer invalidação que acerta em L2 precisa ser enviada para L2, onde fará com que o bloco seja invalidado, se existir.

A dificuldade é o que acontece quando os tamanhos de bloco de L1 e L2 são diferentes. A escolha de tamanhos de bloco diferentes é bastante razoável, pois L2 será muito maior e terá um componente de latência maior em sua penalidade de falta e, assim, desejará usar um tamanho de bloco maior. O que acontece com nossa imposição "automática" de inclusão

quando os tamanhos de bloco diferem? Um bloco em L2 representa múltiplos blocos em L1, e uma falta em L2 causa uma substituição de dados equivalente a múltiplos blocos L1. Por exemplo, se o tamanho de bloco de L2 for quatro vezes o de L1, então uma falta em L2 substituirá o equivalente a quatro blocos L1. Vamos considerar um exemplo detalhado.

Exemplo Suponha que L2 tenha um bloco quatro vezes maior que o de L1. Mostre como uma falta por um endereço que causa substituição em L1 e L2 pode levar à violação da propriedade de inclusão.

Resposta Considere que L1 e L2 sejam mapeados diretamente e que o tamanho de bloco de L1 seja b bytes e o tamanho de bloco de L2 seja $4b$ bytes. Suponha que L1 contenha dois blocos com endereços iniciais, x e $x + b$, e que x mod $4b = 0$, significando que x também é o endereço inicial de um bloco em L2; então, esse único bloco em L2 contém os blocos de L1 x, $x + b$, $x + 2b$ e $x + 3b$. Suponha que o processador gere uma referência ao bloco y que seja mapeada para o bloco contendo x nas duas caches e, portanto, perca. Como L2 gerou uma falta, ele busca $4b$ bytes e substitui o bloco contendo x, $x + b$, $x + 2b$ e $x + 3b$, enquanto L1 apanha b bytes e substitui o bloco contendo x. Como L1 ainda contém $x + b$, mas L2 não, a propriedade de inclusão não permanece.

Para manter a inclusão com múltiplos tamanhos de bloco, temos que sondar os níveis mais altos da hierarquia quando uma substituição for feita no nível inferior para garantir que quaisquer palavras substituídas no nível mais baixo sejam invalidadas nas caches de nível mais alto; diferentes níveis de associatividade criam o mesmo tipo de problema. Em 2011, os projetistas pareciam estar divididos na imposição da inclusão. Baer e Wang (1988) descrevem as vantagens e os desafios da inclusão com detalhes. O Intel i7 usa inclusão para L3, o que significa que L3 sempre inclui todo o conteúdo de L2 e L1. Isso permite a eles implementar um esquema direto de diretório em L3 e minimizar a interface de snooping em L1 e L2 para as circunstâncias em que o diretório indica que L1 ou L2 tem uma cópia na cache. O AMD Opteron, em contraste, torna o L2 inclusivo de L1, mas não tem tal restrição para L3. Eles usam um protocolo snooping, mas só precisa monitorar L2, a menos que haja um acerto, e nesse caso um snoop é enviado para L1.

Ganho de desempenho do uso de multiprocessamento e multithreading

Nesta seção, examinaremos dois estudos diferentes da eficácia de usar multithreading em um processador multicore. Vamos retomar esse tópico na próxima seção, quando examinarmos o desempenho do Intel i7. Nossos dois estudos se baseiam no Sun T1, que apresentamos no Capítulo 3, e no processador IBM Power5.

Nós examinamos o desempenho do multicore T1 usando os mesmos três benchmarks orientados a servidor — TPC-C, SPECJBB (o benchmark SPEC Java empresarial) e SPECWeb99 —, que vimos no Capítulo 3. O benchmark SPECWeb99 é executado somente em uma versão de quatro núcleos do T1, porque ele não pode ser escalado para usar todos os 32 threads de um processador de oito núcleos. Os outros dois benchmarks são executados com oito núcleos e quatro threads cada com um total de 32 threads. A Figura 5.25 mostra os CPIs por thread e por núcleo, o CPI efetivo e instruções por clock (IPC) para o T1 de oito núcleos.

Benchmark	CPI por thread	CPI por núcleo	CPI efetivo para oito núcleos	IPC efetivo para oito núcleos
TPC-C	7,2	1,8	0,225	4,4
SPECJBB	5,6	1,40	0,175	5,7
SPECWeb99	6,6	1,65	0,206	4,8

FIGURA 5.25 CPI por thread, CPI por núcleo, o CPI efetivo para oito núcleos e IPC (inverso do CPI) efetivo para o processador Sun T1 de oito núcleos.

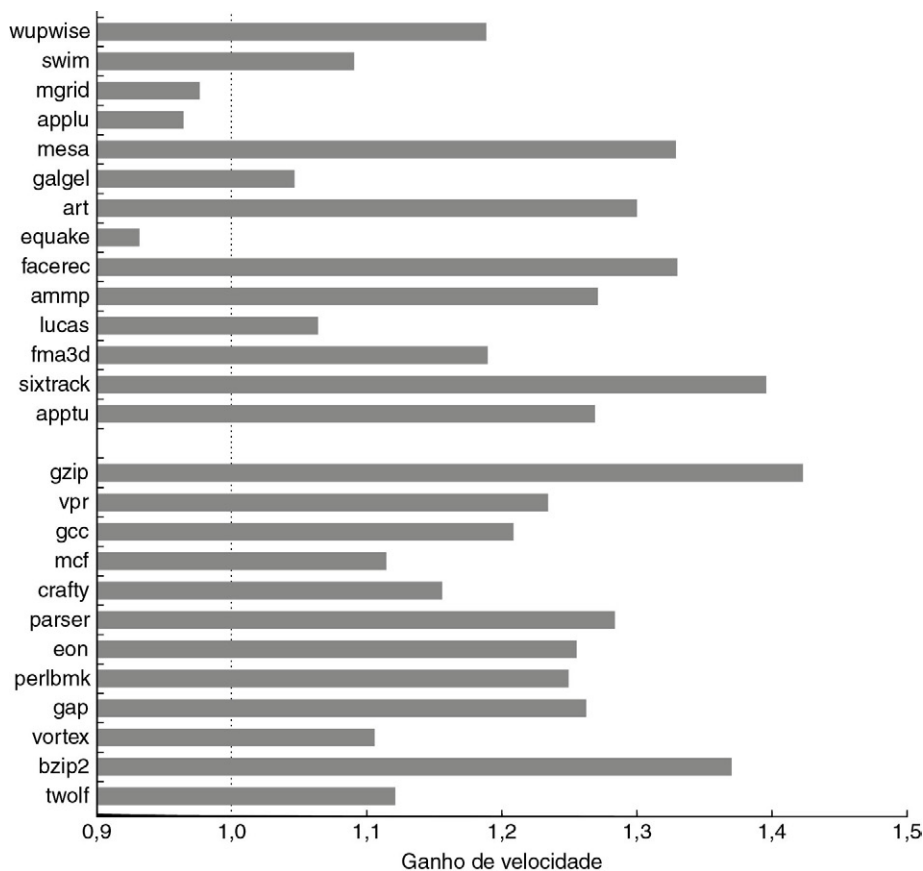



FIGURA 5.26 **Comparação do desempenho SMT e de thread único (ST) no IMB eServer p5 575 de oito processadores.** Observe que o eixo *y* começa em um ganho de velocidade de 0,9, uma perda de desempenho. Somente um processador em cada núcleo do Power5 está ativo, o que deveria melhorar ligeiramente os resultados do SMT ao diminuir a interferência destrutiva no sistema de memória. Os resultados de SMT são obtidos criando 16 threads de usuário, enquanto os resultados de ST usam somente oito threads. Com um único thread por processador, o Power5 é mudado para modo de thread único pelo SO. Esses resultados foram coletados por John McCalpin, da IBM. Como podemos ver a partir dos dados, o desvio-padrão dos resultados para o SPECfpRate é maior do que os para o SPECintRate (0,13 *versus* 0,07), indicando que a melhoria de SMT para programas de PF provavelmente vai variar bastante.

O IBM Power5 é um dual-core que suporta multithreading simultâneo (SMT). Para examinar o desempenho do multithreading em um multiprocessador, foram realizadas medições em um sistema IBM com oito processadores Power5, usando somente um núcleo em cada um. A Figura 5.26 mostra o ganho de velocidade de um multiprocessador Power5 de oito processadores, como descrito na legenda. Na média, o SPECinRate é 1,23 vez mais rápido, enquanto o SPECfpRate é 1,16 vez mais rápido. Observe que alguns benchmarks de ponto flutuante experimentam um leve decréscimo em desempenho no modo SMT, com a redução máxima em ganho de velocidade sendo de 0,93. Embora se possa esperar que o SMT realizasse um trabalho melhor em ocultar as taxas de falta maiores dos benchmarks SPECFP, parece que os limites no sistema de memória são atingidos quando se executam tais benchmarks em modo SMT.

5.8 JUNTANDO TUDO: PROCESSADORES MULTICORE E SEU DESEMPENHO

Em 2011, multicore é um tema de todos os novos processadores. A implementação varia muito, assim como seu suporte a multiprocessadores multichip maiores. Nesta seção, examinaremos o projeto de quatro diferentes processadores multicore e algumas características de desempenho.

Característica	AMD Opteron 8439	IBM Power 7	Intel Xenon 7560	Sun T2
Transistores	904 M	1.200 M	2.300 M	500 M
Potência (nominal)	137 W	140 W	130 W	95 W
Máx. de núcleos/chip	6	8	8	8
Multithreading	Não	SMT	SMT	Fino
Threads/núcleo	1	4	2	8
Despacho de instrução/clock	3 a partir de um thread	6 a partir de um thread	4 a partir de um thread	2 a partir de 2 threads
Frequência	2,8 GHz	4,1 GHz	2,7 GHz	1,6 GHz
Cache mais externa	L3; 6 MB; compartilhada	L3; 32 MB (usando DRAM embutida); compartilhada ou privada/núcleo	L3; 24 MB; compartilhada	L2; 4 MB; compartilhada
Inclusão	Não, embora L2 seja um superconjunto de L1	Sim, superconjunto de L3	Sim, superconjunto de L3	Sim
Protocolo de coerência multicore	MOESI	MESI estendido com dicas de comportamento e localidade (protocolo de 13 estado)	MESIF	MOESI
Implementação de coerência multicore	Snooping	Diretório em L3	Diretório em L3	Diretório em L2
Suporte de coerência estendida	Até 8 chips de processadores podem ser conectados com HyperTransport em anel, usando diretório ou snooping. O sistema é NUMA.	Até 32 chips de processadores podem ser conectados com os links SMP. Estrutura de diretório distribuída dinamicamente. O acesso à memória é simétrico fora de um chip de 8 núcleos.	Até 8 núcleos de processador podem ser implementados através de Interconnect Quickpath. Suporte para diretórios com lógica externa.	Implementado através de quatro links de coerência por processador que podem ser usados para snooping. Até dois chips conectados diretamente, e até quatro conectados usando ASICs externos.

FIGURA 5.27 Resumo das características de quatro processadores multicore de alto nível recentes (lançamentos de 2010) projetados para servidores.

A tabela inclui as versões com maior número de núcleos desses processadores. Existem versões com menor número de núcleos e maiores taxas de clock para muitos desses processadores. O L3 no IBM Power7 pode ser compartilhado ou particionado em regiões comprovadamente mais rápidas, dedicadas a núcleos individuais. Nós incluímos somente implementações de multicores com chip único.

A Figura 5.27 mostra as principais características de quatro processadores multicore projetados para aplicações de servidor. O Intel Xeon se baseia no mesmo projeto que o i7, mas tem mais núcleos, uma frequência ligeiramente menor (a limitação é a potência) e uma cache L3 maior. O AMD Opteron e Phenom para desktop compartilham o mesmo núcleo básico, enquanto o Sun T2 se relaciona ao Sun T1 que encontramos no Capítulo 3. O Power7 é uma extensão do Power5 com mais núcleos e caches maiores.

Primeiro, nós comparamos o desempenho de escalabilidade de desempenho de três desses processadores multicore (omitindo o AMD Opteron, onde não há dados suficientes disponíveis) quando configurados como multiprocessadores multichip.

Além do modo como esses três microprocessadores diferem na sua ênfase em ILP *versus* TLP, há diferenças significativas nos seus mercados-alvo. Assim, nosso foco recairá menos no desempenho comparativo absoluto e mais na escalabilidade do desempenho, conforme processadores adicionais forem adicionados. Depois que examinarmos esses dados, vamos examinar o desempenho multicore do Intel Core i7 detalhadamente.

Nós mostramos o desempenho para três conjuntos de benchmark: SPECintRate, SPECfpRate e SPECjbb2005. Os benchmarks SPECRate que agrupamos ilustram o desempenho desses multiprocessadores para paralelismo em nível de requisição, já que ele é caracterizado pela execução em paralelo e sobreposta de programas independentes. Em particular, nada além dos serviços de sistema é compartilhado. O SPECjbb2005 é um benchmark Java empresarial escalável que modela um sistema cliente/servidor em três partes, com foco no servidor, e é similar ao benchmark usado em SPECPower, que examinamos no Capítulo 1. O benchmark utiliza as implementações da Máquina Virtual Java, mas em tempo de compilador, garbage collection, threads e alguns aspectos do sistema operacional. Ele também testa a escalabilidade dos sistemas multiprocessador.

A Figura 5.28 mostra o desempenho dos benchmarks de CPU SPECRate conforme o aumento do número de núcleos. Um ganho de velocidade quase linear é atingido conforme o aumento do número de chips de processador e, portanto, do número de núcleos.

A Figura 5.29 mostra dados similares para o benchmark SPECjbb2005. As trocas entre explorar mais ILP e se concentrar somente em TLP são complexas e fortemente dependentes da carga de trabalho. O SPECjbb2005 é uma carga de trabalho que aumenta de escala conforme processadores extras são adicionados, mantendo constante o tempo, e não o tamanho do problema. Nesse caso, parece haver amplo paralelismo para obter ganho de

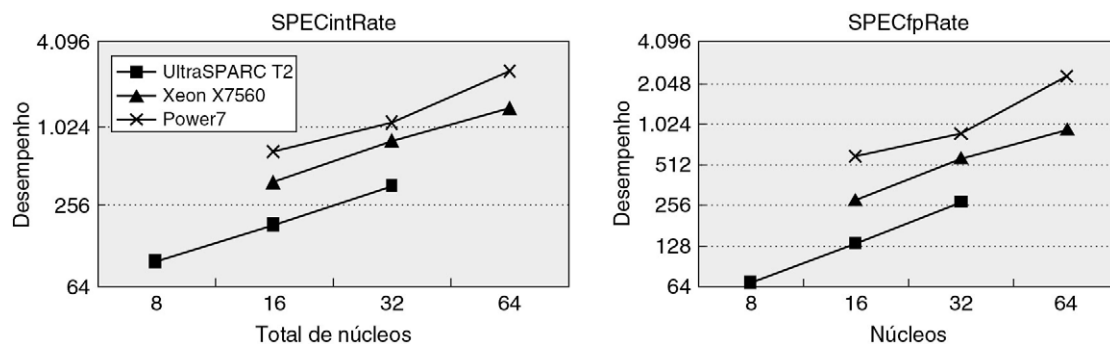



FIGURA 5.28 **O desempenho nos benchmarks SPECRate para três processadores multicore conforme o aumento do número de chips do processador.**
Observe que nesse benchmark altamente paralelo é alcançado um ganho de velocidade quase linear. Esses gráficos estão em escala logarítmica, então o ganho de velocidade linear é uma linha reta.

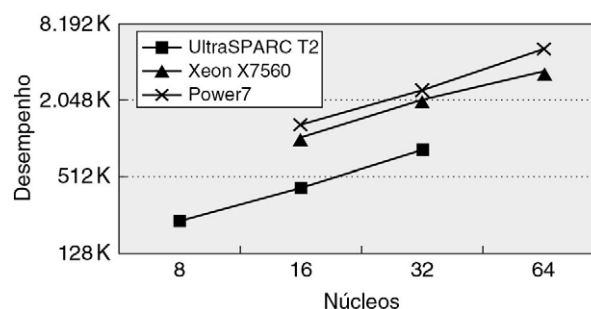



FIGURA 5.29 **O desempenho nos benchmarks SPECjbb2005 para três processadores multicore conforme o aumento do número de chips do processador.**
Observe que nesse benchmark paralelo é alcançado um ganho de velocidade quase linear.

velocidade linear através de 64 núcleos. Vamos retornar a esse tópico nos comentários finais, mas primeiro vamos dar uma olhada melhor no desempenho do Intel Core i7 em modo de chip único com quatro núcleos.

Desempenho e eficiência energética do Intel Core i7 multicore

Nesta seção, vamos examinar o desempenho do i7 nos mesmos dois grupos de benchmarks que consideramos no Capítulo 3: os benchmarks Java paralelos e os benchmarks PARSEC paralelos (descritos em detalhes na Figura 3.34, na página 200). Primeiro, examinamos o desempenho e o escalonamento multicore *versus* um núcleo único sem uso de SMT. Então, combinamos a capacidade multicore e SMT. Todos os dados desta seção, como os da avaliação anterior do SMT do i7 (Cap. 3, Seção 3.13), vêm de Esmaeilzadeh *et al*. (2011). O conjunto de dados é o mesmo usado antes (Figura 3.34, na página 200), exceto pelo fato de que os bechmarks Java tradebeans e pjbb2005 foram removidos (deixando somente os cinco benchmarks Java escaláveis). O tradebeans e pjbb2005 nunca atingem ganho de velocidade acima de 1,55, mesmo com quatro núcleos e um total de oito threads, e por isso não são apropriados para a avaliação de mais núcleos.

A Figura 5.30 mostra o ganho de velocidade e eficiência energética dos benchmarks Java e PARSEC sem o uso de SMT. Mostrar a eficiência energética significa que estamos traçando a razão entre a energia consumida pela execução de dois ou quatro núcleos e a energia consumida pela execução do núcleo único. Maior eficiência energética é melhor, com um valor de 1,0 sendo o ponto de equilíbrio. Os núcleos não utilizados em todos os casos estavam em modo repouso profundo, o que minimizou seu consumo de energia essencialmente desligando-os. Ao comparar os dados para os benchmarks de núcleo único e múltiplos núcleos, é importante nos lembrar de que o custo energético total da cache L3 e da interface de memória é pago no caso do núcleo único (assim como no multicore). Esse fato aumenta a probabilidade de que o consumo de energia vai melhorar

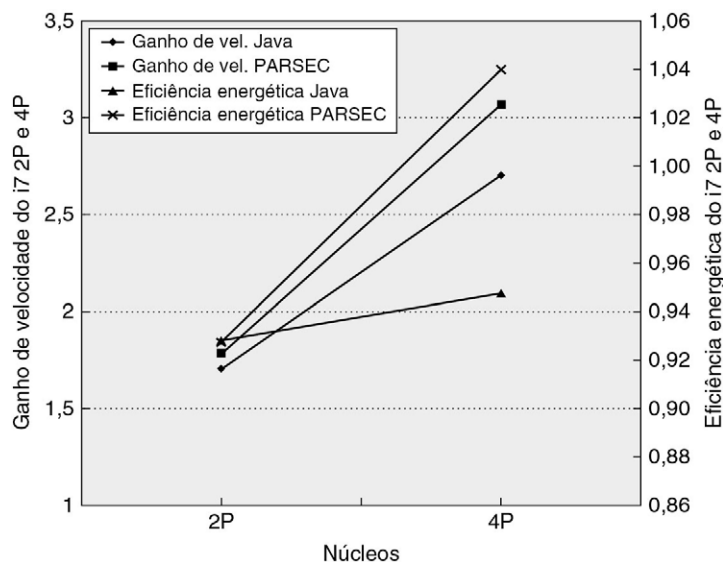



FIGURA 5.30 Esse gráfico mostra o ganho de velocidade para execuções de dois e quatro núcleos das cargas de trabalho Java e PARSEC sem SMT.

Esses dados foram coletados por Esmaeilzadeh *et al*. (2011) usando o mesmo setup descrito no Capítulo 3. O Turbo Boost está desligado. O ganho de velocidade e eficiência energética é resumido usando média harmônica, implicando uma carga de trabalho em que o tempo total executando cada benchmark 2p é equivalente.

para aplicações que podem ser razoavelmente escaladas. A média harmônica é usada para resumir resultados com a implicação descrita na legenda.

Como mostra a Figura 5.30, os benchmarks PARSEC obtêm melhor ganho de velocidade do que os benchmarks Java, atingindo 76% de eficiência no ganho de velocidade (ou seja, ganho de velocidade real dividido pelo número de processadores) em quatro núcleos, enquanto os benchmarks Java atingem 67% de eficiência de ganho de velocidade em quatro núcleos. Embora essa observação esteja clara a partir dos dados apresentados, é difícil analisar por que essa diferença existe. É bem possível, por exemplo, que os efeitos da lei de Amdahl tenham reduzido o ganho de velocidade para a carga de trabalho Java. Além disso, a interação entre a arquitetura do processador e a aplicação, que afeta pontos como o custo de sincronização ou de comunicação, também pode ter influência. Em particular, aplicações bem paralelizadas, como aquelas em PARSEC, às vezes se beneficiam de uma razão vantajosa entre computação e comunicação, que reduz a dependência dos custos de comunicação (Apêndice I).

Essas diferenças em ganho de velocidade se traduzem em diferenças em eficiência energética. Por exemplo, os benchmarks PARSEC realmente melhoram ligeiramente a eficiência energética na versão de núcleo único. Esse resultado pode ser afetado significativamente pelo fato de que a cache L3 é usada com mais eficiência nas execuções multicore do que no caso de núcleo único e de que o custo energético é idêntico nos dois casos. Assim, para os benchmarks PARSEC, a abordagem multicore atinge o que os projetistas esperavam quando mudaram de um projeto focado em ILP para um projeto multicore. Especificamente, ela escala o desempenho com a mesma — ou maior — velocidade com que escala a potência, resultando em eficiência energética constante ou até mesmo melhorada. No caso do Java, vemos que nem as execuções de dois ou quatro núcleos se igualam em eficiência energética, devido aos níveis menores de ganho de velocidade da carga de trabalho Java (embora a eficiência energética para a execução 2p seja a mesma do PARSEC!). A eficiência energética, no caso Java com quatro núcleos, é razoavelmente alta (0,94). É provável que um processador centrado em ILP precisasse de *muito mais* potência para atingir um ganho de velocidade comparável nas cargas de trabalho PARSEC ou Java. Assim, a abordagem centrada em TLP também é melhor do que a abordagem centrada em ILP para melhorar o desempenho para essas aplicações.

Reunindo multicore e SMT

Por fim, consideramos a combinação de multicore e multithreading medindo os dois conjuntos de benchmarks para dois ou quatro processadores e um a dois threads (um total de quatro pontos de dados e até oito threads). A Figura 5.31 mostra o ganho de velocidade e eficiência energética obtida no Intel i7 quando o número de processadores é de dois ou quatro e o SMT é ou não empregado, usando média harmônica para resumir os dois conjuntos de benchmarks. Obviamente, o SMT pode aumentar o desempenho até que haja suficiente paralelismo em nível de thread disponível mesmo na situação multicore. Por exemplo, no caso com quatro núcleos, sem SMT, as eficiências de ganho de velocidade foram de 67% e 76% para Java e PARSEC, respectivamente. Com SMT em quatro núcleos, essas taxas foram impressionantes 83% e 97%!

A eficiência energética apresenta um quadro ligeiramente diferente. No PARSEC, o ganho de velocidade é essencialmente linear para o caso SMT com quatro núcleos (oito threads), e a potência aumenta mais lentamente, resultando em uma eficiência energética de 1,1 para esse caso. A situação Java é mais complexa. A eficiência energética tem um pico na execução SMT (quatro threads) com dois núcleos em 0,97 e cai para 0,89 na execução SMT (oito threads) com quatro núcleos. Parece muito provável que os benchmarks Java encontrem efeitos da lei de Amdahl quando mais de quatro threads são aplicados. Como alguns

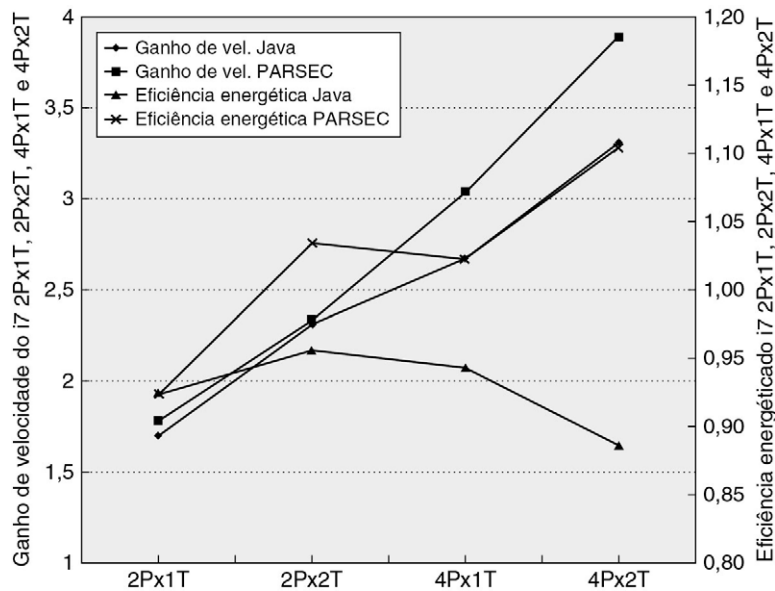



FIGURA 5.31 Esse gráfico mostra o ganho de velocidade para execuções de dois e quatro núcleos das cargas de trabalho paralelas Java e PARSEC com e sem SMT.
Lembre-se de que esses resultados variam em número de threads de dois a oito, refletindo efeitos de arquitetura e características de aplicação. A média harmônica é usada para resumir os resultados, como discutido na legenda da Figura 5.30.

arquitetos observaram, o multicore transfere mais responsabilidade pelo desempenho (e, portanto, mais eficiência energética) para o programador, e os resultados para a carga de trabalho Java certamente suportam isso.

5.9 FALÁCIAS E ARMADILHAS

Dada a falta de maturidade em nosso conhecimento da computação paralela, existem muitas armadilhas ocultas que serão descobertas por projetistas meticulosos ou desafortunados. Dada a grande febre que cercou os multiprocessadores, especialmente os de alto nível, existem muitas falácias. Incluímos uma seleção delas.

Armadilha. *Medir o desempenho de multiprocessadores por ganho de velocidade linear contra tempo de execução.*

Há muito tempo gráficos "tiro de morteiro" — mostrando o desempenho contra o número de processadores, mostrando ganho de velocidade linear, um platô e depois uma queda — têm sido usados para avaliar o sucesso dos processadores paralelos. Embora o ganho de velocidade seja uma faceta de um programa paralelo, não é uma medida direta do desempenho. A primeira questão é a potência dos processadores escalados: um programa que melhora linearmente o desempenho de 100 processadores Intel Atom (o processador de baixo nível usado em netbooks) pode ser mais lento do que a versão executada em um Xeon de oito núcleos. Tenha cuidado especialmente com programas que fazem uso intenso de ponto flutuante; o processamento de elementos sem assistência do hardware pode escalar maravilhosamente bem, mas ter um desempenho coletivo fraco.

Comparar os tempos de execução é justo apenas se você estiver comparando os melhores algoritmos em cada computador. A comparação de código idêntico em dois computadores pode parecer justa, mas não é; o programa paralelo pode ser mais lento em um unipro-

cessador do que uma versão sequencial. Às vezes, o desenvolvimento de um programa paralelo leva a melhorias algorítmicas, de modo que comparar o programa sequencial anteriormente bem conhecido com o código paralelo — que parece justo — não comparará algoritmos equivalentes. Para refletir essa questão, às vezes são usados os nomes *ganho de velocidade relativo* (mesmo programa) e *ganho de velocidade verdadeiro* (melhor programa).

Os resultados que sugerem desempenho *superlinear*, quando um programa em *n* processadores é cerca de *n* vezes mais rápido do que o uniprocessador equivalente, podem indicar que a comparação é injusta, embora existam instâncias em que tenham sido encontrados ganhos de velocidade superlinear "reais". Por exemplo, algumas aplicações científicas obtêm regularmente ganho de velocidade superlinear para pequenos aumentos no número de processadores (dois ou quatro para oito ou 16). Esses resultados normalmente surgem porque as estruturas de dados críticas, que não cabem nas caches agregadas de um multiprocessador com dois ou quatro processadores, cabem na cache agregada de um multiprocessador com oito ou 16 processadores.

Em resumo, comparar o desempenho pela comparação de ganhos de velocidade é no mínimo intrincado e no máximo confuso. A comparação dos ganhos de velocidade para dois multiprocessadores diferentes não nos diz necessariamente algo sobre o desempenho relativo dos multiprocessadores. Até mesmo a comparação de dois algoritmos diferentes no mesmo multiprocessador é intrincada, pois temos que usar o ganho de velocidade verdadeiro em vez do ganho de velocidade relativo para obter uma comparação válida.

Falácia. *A lei de Amdahl não se aplica a computadores paralelos.*

Em 1987, o presidente de uma organização de pesquisa afirmou que a lei de Amdahl (Seção 1.9) tinha sido quebrada por um multiprocessador MIMD. Porém, essa afirmação mal significou que a lei tinha sido alterada para computadores paralelos; a parte negligenciada do programa ainda limitará o desempenho. Para entender a base dos relatos da mídia, vejamos o que Amdahl (1967) disse originalmente:

> Uma conclusão bastante óbvia a que podemos chegar neste ponto é que o esforço gasto para conseguir altas taxas de processamento paralelo é desperdiçado a menos que seja acompanhado de melhorias nas taxas de processamento sequencial quase da mesma magnitude. (página 425)

Uma interpretação da lei foi que, como partes de cada programa precisam ser sequenciais, existe um limite para o número econômico útil de processadores — digamos, 100. Mostrando ganho de velocidade linear com 1.000 processadores, essa interpretação da lei de Amdahl foi refutada.

A base para a afirmação de que a lei de Amdahl tinha sido "superada" foi o uso do *ganho de velocidade escalado*. Os pesquisadores escalaram o benchmark para ter um tamanho de conjunto de dados que fosse 1.000 vezes maior e compararam os tempos de execução do uniprocessador e a execução paralela do benchmark escalado. Para esse algoritmo em particular, a parte sequencial do programa foi constante, independentemente do tamanho da entrada, e o restante foi totalmente paralelo — logo, o ganho de velocidade linear com 1.000 processadores. Como o tempo de execução cresceu mais rapidamente do que o linear, o programa realmente foi executado por mais tempo após a escalada, mesmo com 1.000 processadores.

O ganho de velocidade que assume a escalada da entrada não é igual ao ganho de velocidade verdadeiro, e informá-lo como se fosse é enganoso. Como os benchmarks paralelos normalmente são executados em multiprocessadores de diferentes tamanhos, é importante especificar que tipo de escalada de aplicação é permissível e como essa escalada deve ser feita. Embora apenas escalar o tamanho dos dados com o número de processadores

raramente seja apropriado, assumir um tamanho de problema fixo para um número de processadores muito maior também costuma ser impróprio, pois é provável que, dado um multiprocessador muito maior, os usuários optem por executar uma versão de uma aplicação maior ou mais detalhada. No Apêndice I discutiremos mais esse importante tópico.

Falácia. *Ganhos de velocidade linear são necessários para tornar os multiprocessadores econômicos.*

É bastante reconhecido que um dos principais benefícios da computação paralela é oferecer um "tempo de solução mais curto" do que o uniprocessador mais rápido. Muitas pessoas, porém, sustentam a visão de que os processadores paralelos não podem ser tão econômicos quanto os uniprocessadores, a menos que consigam alcançar o ganho de velocidade linear perfeito. Esse argumento diz que, como o custo do multiprocessador é uma função linear do número de processadores, qualquer coisa menor que o ganho de velocidade linear significa que a razão do custo/desempenho diminui, tornando um processador paralelo menos econômico do que usar um uniprocessador.

O problema com esse argumento é que o custo não é apenas uma função do número de processadores; ele também depende da memória, E/S e overhead do sistema (gabinete, fonte de alimentação, interconexão etc.). Também faz menos sentido na era do multicore, quando há múltiplos processadores por chip.

O efeito de incluir memória no custo do sistema foi apontado por Wood e Hill (1995). Usamos um exemplo baseado nos dados mais recentes, usando os benchmarks TPC-C e SPECRate, mas o argumento também poderia ser feito com uma carga de trabalho de aplicação científica paralela, que provavelmente tornaria o caso ainda mais siginificativo.

A Figura 5.32 mostra o ganho de velocidade para o TPC-C, o SPECintRate e o SPECfpRate em um multiprocessador IBM eServer p5 configurado com 4-64 processadores. A figura

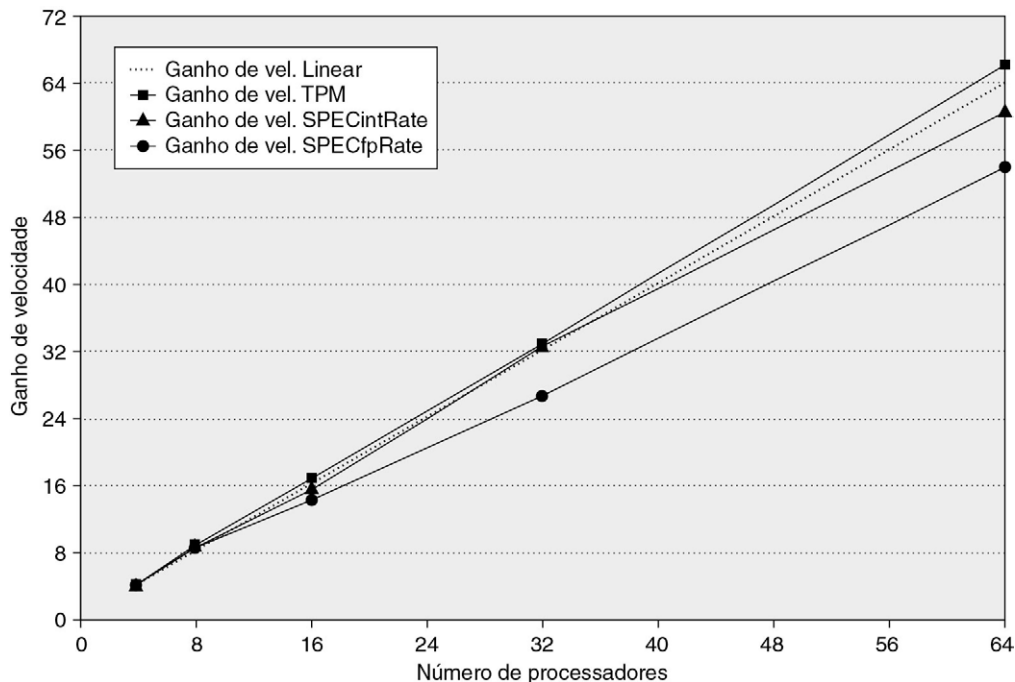



FIGURA 5.32 Ganho de velocidade para três benchmarks em um multiprocessador IBM eServer p5 quando configurado com quatro, oito, 16, 32 e 64 processadores.
A linha tracejada mostra o ganho de velocidade linear.

mostra que somente o TPC-C consegue um ganho de velocidade melhor do que o linear. Para o SPECintRate e o SPECfpRate, o ganho de velocidade é menor do que o linear, mas também o custo, pois, ao contrário do TPC-C, a quantidade de memória principal e de disco exigido é escalada menos que linearmente.

Como mostra a Figura 5.33, número maior de processadores realmente pode ser mais econômico do que a configuração de quatro processadores. Comparando o custo-desempenho dos dois computadores, temos que ter certeza de incluir avaliações precisas do custo total do sistema e de qual desempenho é alcançável. Para muitas aplicações com demandas de memória maiores, tal comparação pode aumentar drasticamente a atratividade do uso de um multiprocessador.

Armadilha. *Não desenvolver o software para tirar proveito de uma arquitetura de multiprocessador ou otimizá-la.*

Existe uma longa história de atraso do software por trás de processadores maciçamente paralelos, possivelmente porque os problemas de software são muito mais difíceis. Daremos um exemplo para mostrar a sutileza das questões, mas existem muitos exemplos que poderíamos escolher!

Um problema frequente é quando o software projetado para um uniprocessador é adaptado para um ambiente de multiprocessador. Por exemplo, em 2000 o sistema operacional

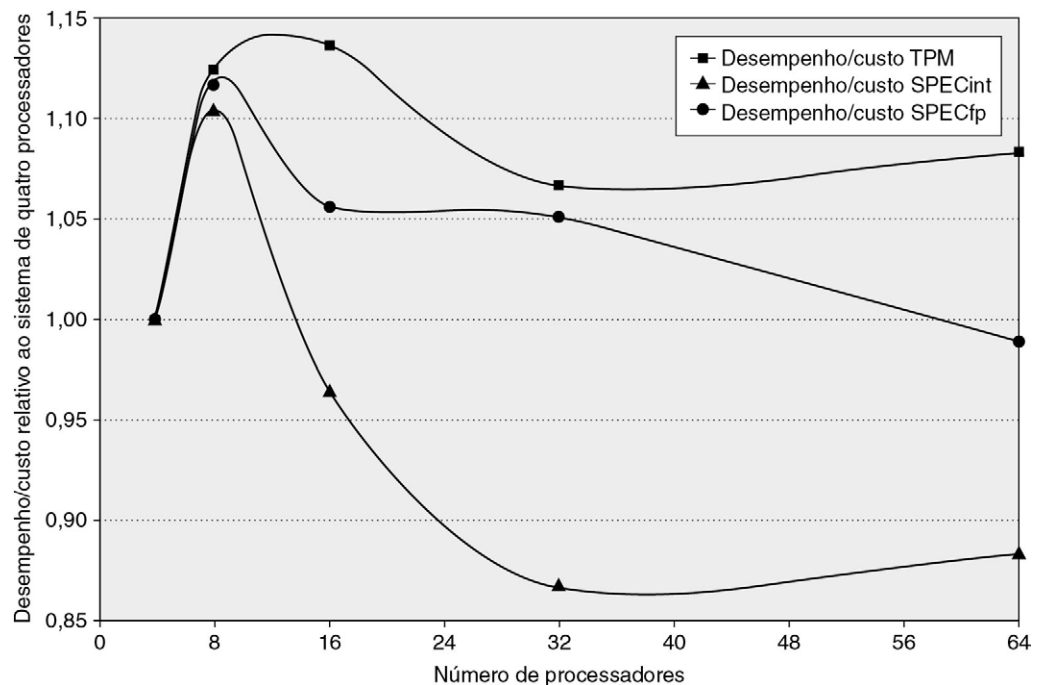



FIGURA 5.33 O desempenho/custo relativo a um sistema de quatro processadores para três benchmarks executados em um multiprocessador IBM eServer p5 contendo 4-64 processadores mostra que a quantidade maior de processadores pode ser tão econômica quanto uma configuração de quatro processadores. Para o TPC-C, as configurações são aquelas usadas nas execuções oficiais, o que significa que o disco e a memória são escalados quase linearmente com o número de processadores, e uma máquina de 64 processadores é aproximadamente o dobro do preço de uma versão de 32 processadores. Ao contrário, o disco e a memória são escalados mais lentamente (embora ainda mais rápido do que o necessário para alcançar o melhor SPECRate em 64 processadores). Em particular, as configurações de disco vão de uma unidade para a versão de quatro processadores até quatro unidades (140 GB) para a versão de 64 processadores. A memória é escalada de 8 GB para o sistema de quatro processadores para 20 GB para o sistema de 64 processadores.

SGI protegeu originalmente a estrutura de dados da tabela de página com um único bloqueio, supondo que a alocação de página é pouco frequente. Em um uniprocessador, isso não representa um problema para o desempenho. Em um multiprocessador, pode se tornar um gargalo de desempenho importante para alguns programas. Considere um programa que usa grande quantidade de páginas que são inicializadas na partida, o que o UNIX faz para páginas estaticamente alocadas. Suponha que o programa seja colocado em paralelo de modo que múltiplos processos aloquem as páginas. Como a alocação de página exige o uso da estrutura de dados da tabela de página, que é bloqueada sempre que está em uso, até mesmo um kernel do SO que permite múltiplos threads no SO usará a serialização se todos os processos tentarem alocar suas páginas ao mesmo tempo (o que é exatamente o que poderíamos esperar no momento da inicialização!).

Essa serialização da tabela de página elimina o paralelismo na inicialização e tem impacto significativo sobre o desempenho paralelo em geral. Esse gargalo de desempenho persiste mesmo sob a multiprogramação. Por exemplo, suponha que dividamos o programa paralelo em processos separados e os executemos, um processo por processador, de modo que não haja compartilhamento entre os processos. (É exatamente isso que um usuário faria, pois acredita que o problema de desempenho deveu-se ao compartilhamento não intencionado ou interferência em sua aplicação.) Infelizmente, o bloqueio ainda serializa todos os processos, de modo que até mesmo o desempenho de multiprogramação é fraco. Essa armadilha indica sutis porém significativos tipos de erros e de desempenho, que podem surgir quando o software é executado nos multiprocessadores. Como muitos outros componentes principais de software, os algoritmos de SO e as estruturas de dados precisam ser repensados em um contexto de multiprocessador. Colocar bloqueios sobre partes menores da tabela de página elimina efetivamente o problema. Existem problemas semelhantes nas estruturas de memória, que aumentam o tráfego de coerência em casos nos quais nenhum compartilhamento está realmente ocorrendo.

Conforme o multicore se tornou o tema dominante em tudo, de desktops a servidores, a falta de investimento adequado em software paralelo se tornou aparente. Dada a falta de foco, provavelmente levará muitos anos antes que os sistemas de software que usamos explorem adequadamente esse crescimento no número de núcleos.

5.10 COMENTÁRIOS FINAIS

Por mais de 30 anos, os pesquisadores e projetistas previram o fim dos uniprocessadores e seu domínio pelos multiprocessadores. Até os primeiros anos deste século, essa previsão foi constantemente refutada. Como vimos no Capítulo 3, os custos para tentar encontrar e explorar mais ILP são proibitivos em eficiência (tanto de área de silício quanto de consumo de energia). Obviamente, os multicores não solucionam o problema de potência, já que eles claramente aumentam tanto o número de transistores como o número de transistores ativos sendo comutados, que são os dois contribuintes dominantes para o consumo de energia.

Entretanto, os multicores realmente mudam o jogo. Ao permitir que núcleos ociosos sejam colocados em modo de economia de energia, alguma melhoria na eficiência energética pode ser alcançada, como mostraram os resultados neste capítulo. E o que é mais importante: o multicore transfere o fardo de manter o processador ocupado dependendo mais do TLP, o qual a aplicação e o programador são responsáveis por identificar, do que do ILP, pela qual o hardware é responsável. Como vimos, essas diferenças aparecem claramente no desempenho de multicore e na eficiência energética dos benchmarks Java em comparação com os benchmarks PARSEC.

Embora o multicore forneça alguma ajuda direta com o desafio da eficiência energética e transfira grande parte do fardo para o sistema de software, ainda existem desafios diferentes e questões não resolvidas. Por exemplo, até agora as tentativas de explorar versões em nível de thread de especulação agressiva tiveram o mesmo destino de suas contrapartes do ILP. Ou seja, os ganhos de desempenho foram modestos e provavelmente são menores que o aumento de consumo energético, então ideias como threads especulativos ou run-ahead de hardware não tiveram sucesso ao serem incorporadas nos processadores. Como na especulação para ILP, a menos que a especulação esteja quase sempre certa, os custos vão exceder os benefícios.

Além dos problemas centrais das linguagens de programação e tecnologia de compilador, os multicores reabriram outra questão há muito tempo pendente sobre arquiteturas de computador: vale a pena considerar processadores heterogêneos? Embora um multicore assim ainda não tenha sido lançado e os multiprocessadores heterogêneos tenham tido sucesso limitado em computadores de objetivo especial ou sistemas embarcados, as possibilidades são muito maiores em um ambiente multicore. Assim como ocorre com muitas questões em multiprocessamento, a resposta provavelmente vai depender dos modelos de software e sistemas de programação. Se os compiladores e sistemas operacionais puderem usar com eficiência os processadores heterogêneos, eles vão se tornar mais populares. No momento, lidar eficientemente com números modestos de núcleos homogêneos está além dos recursos dos compiladores atuais para muitas aplicações, mas os multiprocessadores que têm núcleos heterogêneos com claras diferenças em capacidade funcional e métodos óbvios para decompor uma aplicação estão se tornando mais comuns, incluindo unidades especiais de processamento como GPUs e processadores de mídia. A ênfase na eficiência energética também poderia levar à inclusão de núcleos com razões desempenho/consumo de energia.

Na edição de 1995, concluímos o capítulo com uma análise de duas questões então controvertidas:

1. Que arquitetura os multiprocessadores de escala muito grande baseados em microprocessador usariam?
2. Qual será o papel do multiprocessamento no futuro da arquitetura de microprocessador?

Os anos seguintes responderam em grande parte a essas duas questões.

Como os multiprocessadores de escala muito grande não se tornaram um mercado importante e em crescimento, a única maneira econômica de montá-los em grande escala era usar clusters em que os nós individuais são microprocessadores únicos ou microprocessadores em escala moderada e memória compartilhada (em geral, dois ou quatro multicores), e a tecnologia de interconexão é tecnologia-padrão de rede. Esses clusters, que têm sido escalados para dezenas de milhares de processadores e instalados em "wharehouses" especialmente projetados, são o tema do Capítulo 6.

Só recentemente a resposta para a segunda pergunta tornou-se surpreendentemente clara. O aumento do desempenho futuro dos microprocessadores, pelo menos nos próximos cinco anos, certamente virá da exploração do paralelismo em nível de thread através de processadores multicore, e não da exploração de mais ILP.

Como consequência disso, os núcleos se tornaram os novos blocos de construção dos chips; os fornecedores oferecem uma variedade de chips baseados em um projeto de núcleo usando números de núcleos diferentes e de caches L3. A Figura 5.34, por exemplo, mostra a família de processadores Intel construída usando somente o núcleo Nehalem (usado no Xeon 7560 e no i7)!

Processador	Série	Núcleos	Cache L3	Potência (típica)	Frequência (GHz)	Preço
Xeon	7500	8	18-24 MB	130 W	2-2,3	$2.837-3.692
Xeon	5600	4-6 com/sem SMT	12 MB	40-130 W	1,86-3,33	$440-1.663
Xeon	3400-3500	4 com/sem SMT	8 MB	45-130 W	1,86-3,3	$189-999
Xeon	5500	2-4	4-8 MB	80-130 W	1,86-3,3	$80-1.600
i7	860-975	4	8 MB	82 W-130 W	2,53-3,33	$284-999
i7 móvel	720-970	4	6-8 MB	45-55 W	1,6-2,1	$364-378
i5	750-760	4 sem SMT	8 MB	80 W	2,4-2,8	$196-209
i3	330-350	2 com/sem SMT	3 MB	35 W	2,1-2,3	

FIGURA 5.34 Características para uma faixa de partes Intel baseadas na microarquitetura Nehalem.
Essa tabela ainda omite uma variedade de entradas em cada linha (2 a 8!). O preço é para um pedido de 1.000 unidades.

Nas décadas de 1980 e 1990, com o nascimento e o desenvolvimento do ILP, o software na forma de compiladores de otimização que poderiam explorar o ILP foi a chave do sucesso. De modo semelhante, a bem-sucedida exploração do paralelismo em nível de thread dependerá muito do desenvolvimento de sistemas de software adequados, assim como das contribuições dos arquitetos de computador. Dado o lento progresso no software paralelo nos últimos trinta e poucos anos, é provável que a exploração do paralelismo em nível de thread permaneça bastante desafiadora por muitos anos. Além do mais, os autores acreditam que existe uma oportunidade significativa para melhores arquiteturas multicore. Para projetar, esses arquitetos precisarão de uma disciplina quantitativa de projeto e da capacidade de modelar com precisão dezenas a centenas de núcleos executando trilhões de instruções, incluindo aplicações de grande escala e sistemas operacionais. Sem tal tecnologia e capacidade, os arquitetos darão tiros no escuro. Às vezes você tem sorte, mas na maioria das vezes erra.

5.11 PERSPECTIVAS HISTÓRICAS E REFERÊNCIAS

A Seção L.7 (disponível on-line) examina a história dos multiprocessadores e o processamento paralelo. Dividida tanto pelo período de tempo quanto pela arquitetura, a seção inclui discussões sobre os primeiros multiprocessadores experimentais e alguns dos maiores debates sobre o processamento paralelo. Os avanços recentes também são abordados, e são incluídas referências de leitura adicionais.

ESTUDOS DE CASO COM EXERCÍCIOS POR AMR ZAKY E DAVID A. WOOD

Estudo de caso 1: Multiprocessador multicore de chip único

Conceitos ilustrados por este estudo de caso

- Transições do protocolo de coerência snooping
- Desempenho do protocolo de coerência
- Otimizações do protocolo de coerência
- Sincronismo
- Desempenho dos modelos de consistência de memória

O multiprocessador simples ilustrado na Figura 5.35 representa uma arquitetura de memória simétrica, comumente implementada. Cada processador tem uma única cache privada com a coerência mantida por meio do protocolo de coerência snooping da Figura 5.7. Cada

cache tem mapeamento direto, com quatro blocos e com duas palavras cada bloco. Para simplificar a ilustração, a tag de endereço da cache contém o endereço completo, e cada palavra mostra apenas dois caracteres hexa, com a palavra menos significativa à direita. Os estados de coerência são indicados por M, S e I, de Modificado, Shared (compartilhado) e Inválido.

5.1 [10/10/10/10/10/10/10] <5.2> Para cada parte deste exercício, considere a cache inicial e o estado da memória conforme ilustrados na Figura 5.35. Cada parte deste exercício especifica uma sequência de uma ou mais operações da CPU na forma:

```
P#: <op> <endereço> [<valor>]
```

onde P# designa a CPU (p. ex., `P0`), <`op`> é a operação da CPU (p. ex., read ou write), <`endereço`> indica o endereço da memória e <`valor`> indica a nova palavra a ser atribuída em uma operação de escrita. Trate cada ação a seguir como sendo independentemente aplicada ao estado inicial, conforme dado na Figura 5.35. Qual é o estado resultante (ou seja, estado de coerência, tags e dados) das caches e memória após a ação indicada? Mostre apenas os blocos que mudam; por exemplo, `P0.B0`: (I, 120, 00 01) indica que o bloco `B0` da CPU `P0` tem o estado final de I, tag de 120 e palavras de dados 00 e 01. Além disso, que valor é retornado por operação de leitura?

```
a. [10] <5.2> P0: read 120
b. [10] <5.2> P0: write 120 <-- 80
c. [10] <5.2> P3: write 120 <-- 80
d. [10] <5.2> P1: read 110
e. [10] <5.2> P0: write 108 <-- 48
f. [10] <5.2> P0: write 130 <-- 78
g. [10] <5.2> P3: write 130 <-- 78
```

5.2 [20/20/20/20] <5.3> O desempenho de um multiprocessador com cache com coerênncia por snooping depende de muitas questões de implementação que determinam a rapidez com que uma cache responde com dados em um bloqueio exclusivo ou em estado *M*. Em algumas implementações, uma falta de leitura da CPU a um bloco da cache que é exclusivo na cache de outro processador é mais rápida do que uma falta a um bloco na memória. Isso porque as caches são menores e, portanto, mais rápidas do que a memória principal. Reciprocamente, em algumas implementações, as faltas satisfeitas pela memória são mais rápidas do que aquelas satisfeitas pelas caches, porque as caches geralmente são otimizadas para a "frente" ou referências da CPU, em vez de para "trás" ou acessos por snooping. Para o multiprocessador ilustrado na Figura 5.35, considere a execução de uma sequência de operações em uma única CPU, onde

- Acertos de leitura e escrita de CPU não geram ciclos de stall.
- Faltas de leitura e escrita de CPU geram ciclos de stall $N_{\text{memória}}$ e N_{cache}, se satisfeitas pela memória e pela cache, respectivamente.
- Acertos de escrita de CPU que geram uma invalidação incorrem em $N_{\text{invalidação}}$ ciclos de stall.
- Um write-back de um bloco, seja devido a um conflito, seja devido à solicitação de outro processador a um bloco exclusivo, incorre em $N_{\text{write-back}}$ ciclos de stall adicionais.

Considere duas implementações com as diferentes características de desempenho resumidas na Figura 5.36. Considere a sequência de operações a seguir,

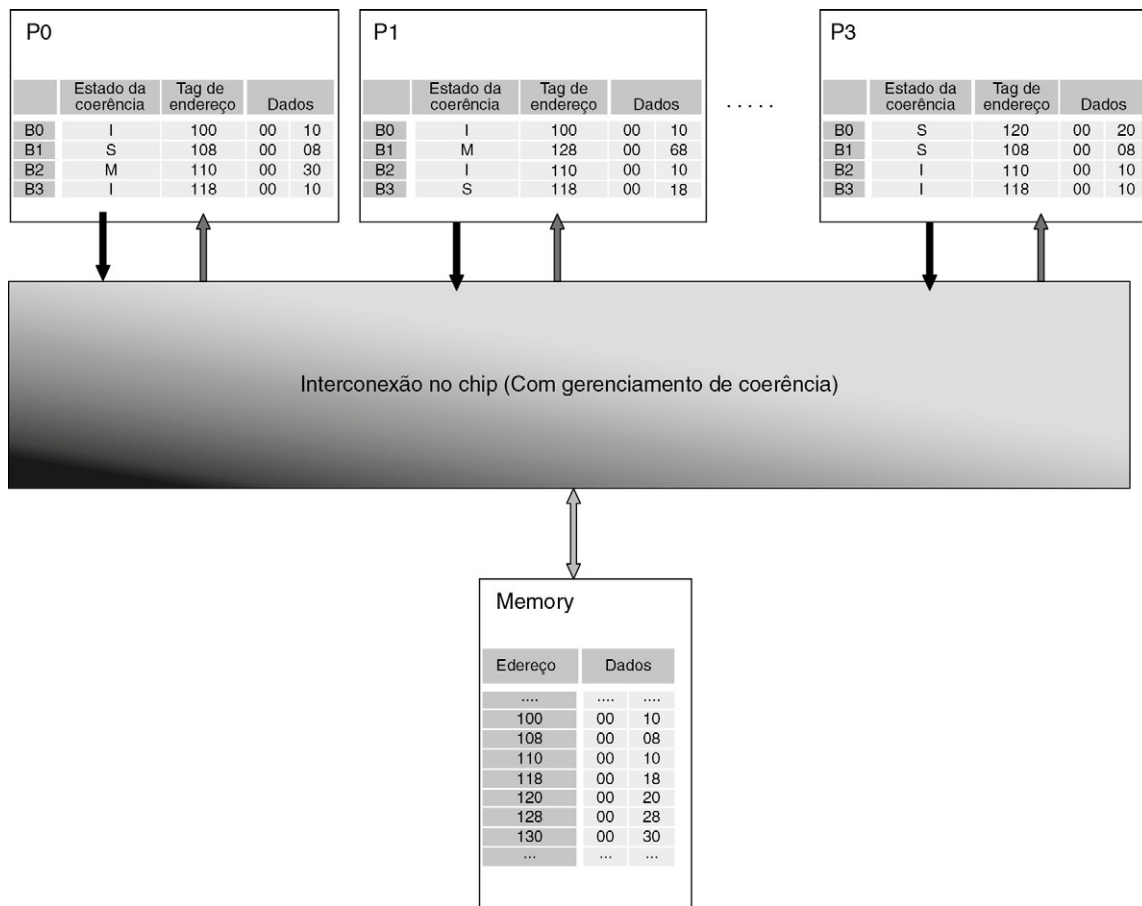



FIGURA 5.35 Multiprocessador multicore (ponto a ponto).

Parâmetro	Implementação 1	Implementação 2
$N_{memória}$	100	100
N_{cache}	40	130
$N_{invalidate}$	15	15
$N_{write\text{-}back}$	10	10

FIGURA 5.36 Latências de coerência snooping.

considerando o estado da cache inicial na Figura 5.35. Para simplificar, considere que a segunda operação começa depois que a primeira termina (embora estejam em processadores diferentes):

```
P1: read 110
P3: read 110
```

Para a implementação 1, a primeira leitura gera 50 ciclos de stall, pois a leitura é satisfeita pela cache de P0. P1 aguarda por 40 ciclos enquanto espera o bloco, e P0 por 10 ciclos enquanto escreve o bloco de volta à memória em resposta à solicitação de P1. Assim, a segunda leitura por P3 gera 100 ciclos de stall, pois sua falta é

resolvida pela memória. Então, essa sequência gera um total de 150 ciclos de stall. Para as sequências de operações a seguir, quantos ciclos de stall são gerados por implementação?

```
a.    [20] <5.3>  P0: read 120
                  P0: read 128
                  P0: read 130
b.    [20] <5.3>  P0: read 100
                  P0: write 108 <-- 48
                  P0: write 130 <-- 78
c.    [20] <5.3>  P1: read 120
                  P1: read 128
                  P1: read 130
d.    [20] <5.3>  P1: read 100
                  P1: write 108 <-- 48
                  P1: write 130 <-- 78
```

5.3 [20] <5.2> Muitos protocolos de coerência de snooping possuem estados adicionais, transições de estado ou transações de barramento para reduzir o overhead de manutenção da coerência de cache. Na implementação 1 do Exercício 5.2, as faltas estão gerando menos ciclos de stall quando são fornecidos pela cache do que quando são fornecidas pela memória. Alguns protocolos de coerência tentam melhorar o desempenho aumentando a frequência desse caso. Uma otimização de protocolo comum consiste em introduzir um estado Owned (normalmente, indicado por 0). O estado Owned se comporta como o estado Shared, visto que os nós só podem ler blocos Owned. Mas ele se comporta como o estado Modificado, visto que os nós precisam fornecer dados nas faltas de leitura e escrita de outros nós para os blocos Owned. Uma falta de leitura a um bloco nos estados Modificado ou Owned fornece dados para o nó solicitante e transições para o estado Owned. Uma falta de escrita para um bloco em qualquer estado, Modificado ou Owned, fornece dados ao nó solicitante e transições para o estado Inválido. Esse protocolo MOSI otimizado só atualiza a memória quando um nó substitui um bloco no estado Modificado ou Owned. Desenhe novos diagramas de protocolo com o estado adicional e suas transições.

5.4 [20/20/20/20] <5.2> Para as sequências de código a seguir e os parâmetros de temporização para as duas implementações na Figura 5.36, calcule os ciclos de stall totais para o protocolo básico MSI e o protocolo MOSI otimizado no Exercício 5.3. Considere que as transições de estado que não exigem transações do barramento não incorrem em ciclos adicionais de stall.

```
a.    [20] <5.2>  P0: read 110
                  P3: read 110
                  P0: read 110
b.    [20] <5.2>  P1: read 120
                  P3: read 120
                  P0: read 120
c.    [20] <5.2>  P0: write 120 <-- 80
                  P3: read 120
                  P0: read 120
d.    [20] <5.2>  P0: write 108 <-- 88
                  P3: read 108
                  P0: write 108 <-- 98
```

5.5 [20] <5.2> Algumas aplicações leem um conjunto de dados grande primeiro, depois modificam a maior parte ou tudo. O protocolo básico de coerência

MSI primeiro apanhará todos os blocos de cache no estado Shared e depois será forçado a realizar uma operação de invalidação para fazer a atualização para o estado Modificado. O atraso adicional tem um impacto significativo sobre algumas cargas de trabalho. Uma otimização adicional do protocolo elimina a necessidade de fazer a atualização dos blocos que são lidos e mais tarde escritos por um único processador. Essa otimização acrescenta um estado Exclusivo (E) ao protocolo, indicando que nenhum outro nó tem uma cópia do bloco, mas ainda não foi modificado. Um bloco da cache entra no estado Exclusivo quando uma falta de leitura é atendida pela memória e nenhum outro nó tem uma cópia válida. Leituras e escritas da CPU nesse bloco prosseguem sem mais tráfego no barramento, mas as escritas da CPU fazem com que o estado de coerência passe para Modificado. Exclusivo difere de Modificado, porque o nó pode substituir silenciosamente os blocos Exclusivos (enquanto os blocos Modificados precisam ser escritos de volta à memória). Além disso, uma falta de leitura para um bloco Exclusivo resulta em uma transição para Shared, mas não exige que o nó responda com dados (pois a memória possui uma cópia atualizada). Desenhe novos diagramas de protocolo para um protocolo MESI que acrescenta o estado Exclusivo e transições para os estados Modificado, Compartilhado e Inválido do protocolo MSI.

5.6 [20/20/20/20/20] <5.2> Considere o conteúdo da cache da Figura 5.35 e a implementação 1 na Figura 5.36. Quais são os ciclos de stall totais para as sequências de código a seguir com o protocolo básico MSI e o novo protocolo MESI no Exercício 5.5? Considere que as transições de estados que não exigem transações do barramento não incorrem em ciclos de stall adicionais.

```
a. [20] <5.2>     P0: read 100
                  P0: write 100 <-- 40
b. [20] <5.2>     P0: read 120
                  P0: write 120 <-- 60
c. [20] <5.2>     P0: read 100
                  P0: read 120
d. [20] <5.2>     P0: read 100
                  P1: write 100 <-- 60
e. [20] <5.2>     P0: read 100
                  P0: write 100 <-- 60
                  P1: write 100 <-- 40
```

5.7 [20/20/20/20] <5.5> O spin lock é o mecanismo de sincronismo mais simples possível na maioria das máquinas comerciais de memória compartilhada. Esse spin lock conta com o primitivo de troca para carregar atomicamente o valor antigo e armazenar um novo valor. A rotina de bloqueio realiza a operação de troca repetidamente até que encontre o bloqueio desbloqueado (ou seja, o valor retornado é 0).

```
DADDUI R2,R0,#1
lockit:     EXCH R2,0(R1)
            BNEZ R2, lockit
```

O desbloqueio de um spin lock simplesmente exige um armazenamento do valor 0.

```
unlock: SW R0,0(R1)
```

Conforme vimos na Seção 5.5, o spin lock mais otimizado emprega coerência de cache e usa um load para verificar o bloqueio, permitindo que ele "gire" com uma variável compartilhada na cache.

```
lockit:   LD      R2, 0(R1)
          BNEZ    R2, lockit
          DADDUI  R2,R0,#1
          EXCH    R2,0(R1)
          BNEZ    R2, lockit
```

Suponha que os processadores P0, P1 e P3 estejam tentando adquirir um bloqueio no endereço 0 × 100 (ou seja, o registrador R1 mantém o valor 0 × 100). Considere o conteúdo da cache da Figura 5.35 e os parâmetros de temporização da implementação 1 na Figura 5.36. Para simplificar, considere que as seções críticas utilizam 1.000 ciclos.

a. [20] <5.5> Usando o spin lock simples, determine *aproximadamente* quantos ciclos de stall da memória cada processador incorre antes de adquirir o bloqueio.

b. [20] <5.5> Usando o spin lock otimizado, determine *aproximadamente* quantos ciclos de stall da memória cada processador incorre antes de adquirir o bloqueio.

c. [20] <5.5> Usando o spin lock simples, *aproximadamente* quantas transações de barramento ocorrem?

d. [20] <5.5> Usando o spin lock otimizado, *aproximadamente* quantas transações de barramento ocorrem?

5.8 [20/20/20/20] <5.6> A consistência sequencial (SC) requer que todas as leituras e gravações pareçam ter sido executadas em alguma ordem total. Isso pode requerer que o processador paralise (stall) em certos casos antes de confirmar uma instrução de escrita ou leitura. Considere a sequência de código a seguir:

```
write A
read B
```

onde write A resulta em uma falta de cache e read B resulta em um acerto de cache. Sob SC, o processador deve paralisar read B até que ele possa ordenar (e assim realizar) write A. Implementações simples de SC vão paralisar o processador até que a cache receba os dados e possa realizar a gravação. Modelos de consistência mais fracos relaxam as restrições de ordenação sobre leituras e escritas, reduzindo os casos que o processador deve paralisar. O modelo de consistência Total Store Order (TSO) requer que todas as escritas pareçam ocorrer em uma ordem total, mas permite que as leituras de um processador passem suas próprias escritas. Isso permite aos processadores implementar buffers de escrita que mantêm escritas confirmadas que ainda não tenham sido ordenadas em relação às escritas de outros processadores. Permite-se que leituras passem (e potencialmente ignorem) o buffer de escrita em TSO (o que eles não poderiam fazer em SC). Suponha que uma operação de memória possa ser realizada por ciclo e que as operações que gerem acertos na cache ou que possam ser resolvidas pelo buffer de escrita não introduzem ciclos de stall. As operações que têm faltas incorrem nas latências listadas na Figura 5.36.
Considere o conteúdo de cache na Figura 5.35. Quantos ciclos de stall ocorrem *antes* de cada operação para os modelos de consistência SC e TSO?

```
a. [20] <5.6>     P0: write 110 <-- 80
                  P0: read 108
b. [20] <5.6>     P0: write 100 <-- 80
                  P0: read 108
c. [20] <5.6>     P0: write 110 <-- 80
                  P0: write 100 <-- 90
d. [20] <5.6>     P0: write 100 <-- 80
                  P0: write 110 <-- 90
```

Estudo de caso 2: Coerência simples baseada em diretório

Conceitos ilustrados por este estudo de caso

- Transições do protocolo de coerência de diretório
- Desempenho do protocolo de coerência
- Otimizações do protocolo de coerência

Considere o sistema de memória compartilhada distribuída, ilustrado na Figura 5.37. Ele consiste em dois chips de quatro núcleos. O processador em cada chip compartilha uma cache L2 (L2$), e os dois chips são conectados através de uma interconexão ponto a ponto. A memória do sistema é distribuída através dos dois chips. A Figura 5.38 mostra, em detalhe, parte desse sistema. Pi,j denota processador *i* no chip *j*. Cada processador possui uma única cache L1 de mapeamento direto, que mantém dois blocos de duas palavras cada. Cada chip tem uma única cache L2 de mapeamento direto que contém dois blocos, cada qual com duas palavras. Para simplificar a ilustração, a tag de endereço da cache contém o endereço completo e cada palavra mostra apenas dois caracteres hexa, com a

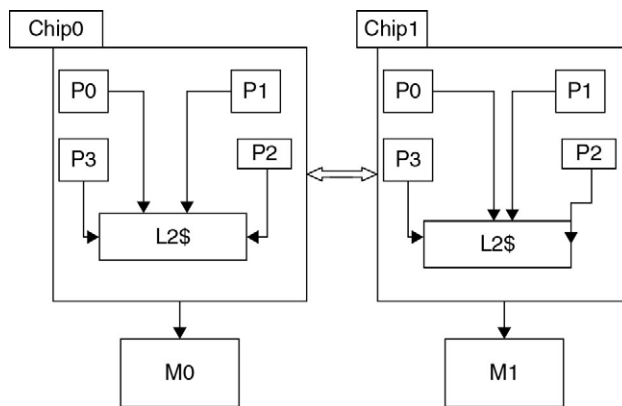



FIGURA 5.37 Multiprocessador multichip, multicore, com DSM.

P0,0

	Estado da coerência	Tag de endereço	Dados	
B0	M	100	00	10
B1	S	108	00	08

P0,1

	Estado da coerência	Tag de endereço	Dados	
B0	M	130	00	68
B1	S	118	00	18

. . . . . .

P3,1

	Estado da coerência	Tag de endereço	Dados	
B0	S	120	00	20
B1	S	108	00	08

L2$, 0

Endereço	Estado	Prop./ Compartilh.	Tag de endereço	Dados	
B0	DM	P0,1	100	00	10
B1	DS	P0,0; E	108	00	08
B2	DM	P1,0	130	00	68
B3	DS	P1,0	118	00	18

L2$,1

Endereço	Estado	Prop./ Compartilh.	Tag de endereço	Dados	
B0	DS	P3,1	120	00	20
B1	DS	P3,1; E	108	00	08
B2	DI	-	-	00	10
B3	DI	-	-	00	20

M0

Endereço	Estado	Prop./ Compartilh.	Dados	
100	DM	C0	00	10
108	DS	C0, C1	00	08
110	DI	-	00	10
118	DS	C0	00	18

M1

Endereço	Estado	Prop./ Compartilh.	Dados	
120	DS	C1	00	20
128	DI	-	00	28
130	DM	C0	00	68
138	DI	-	00	96

FIGURA 5.38 Estado de cache e memória no multiprocessador multichip, multicore.

palavra menos significativa à direita. Os estados da cache L1 são indicados com M, S e I, de Modificado, Shared (compartilhado) e Inválido. As caches L2 e as memórias têm diretórios. Os estados do diretório são indicados por DM, DS e DI, de Diretório Modificado, Diretório Shared e Diretório Inválido. O protocolo de diretório simples é descrito nas Figuras 5.22 e 5.23. O diretório de L2 lista os sharers/owners locais e, além disso, registra se uma linha é compartilhada externamente em outros chips. Por exemplo, P1,0;E denota que uma linha é compartilhada pelo processador local, P1,0, e é compartilhada externamente em algum outro chip. O diretório da memória tem uma lista dos chips sharers/owners de uma linha. Por exemplo, C0,C1 denota que uma linha é compartilhada nos chips 0 e 1.

5.9 [10/10/10/10/15/15/15/15] <5.4> Para cada parte deste exercício, considere o estado inicial da cache e da memória na Figura 5.38. Cada parte especifica uma sequência de uma ou mais operações de CPU na forma:

```
P#: <op> <endereço> [ <-- <valor> ]
```

onde P# designa a CPU (p. ex., P0,0), <op> é a operação da CPU (p. ex., read ou write), <endereço> indica o endereço da memória e <valor> indica a nova palavra a ser atribuída em uma operação de escrita. Qual é o estado final (ou seja, estado de coerência, tags e dados) das caches e da memória após a sequência indicada de operações da CPU ter sido concluída? Além disso, que valor é retornado por operação read?

```
a. [10] <5.4>     P0,0: read 100
b. [10] <5.4>     P0,0: read 128
c. [10] <5.4>     P0,0: write 128 <-- 78
d. [10] <5.4>     P0,0: read 120
e. [15] <5.4>     P0,0: read 120
                  P1,0: read 120
f. [15] <5.4>     P0,0: read 120
                  P1,0: write 120 <-- 80
g. [15] <5.4>     P0,0: write 120 <-- 80
                  P1,0: read 120
h. [15] <5.4>     P0,0: write 120 <-- 80
                  P1,0: write 120 <-- 90
```

5.10 [10/10/10/10] <5.4> Os protocolos de diretório são mais escaláveis do que os protocolos snooping, pois enviam mensagens explícitas de solicitação e invalidação para os nós que possuem cópias de um bloco, enquanto os protocolos snooping enviam todas as solicitações e invalidações por broadcast a todos os nós. Considere o sistema de 16 processadores ilustrado na Figura 5.37 e suponha que todas as caches não mostradas possuem blocos inválidos. Para cada uma das sequências a seguir, identifique quais nós recebem cada solicitação e invalidação.

```
a. [10] <5.4> P0,0: write 100 <-- 80
b. [10] <5.4> P0,0: write 108 <-- 88
c. [10] <5.4> P0,0: write 118 <-- 90
d. [10] <5.4> P1,0: write 128 <-- 98
```

5.11 [25] <5.4> O Exercício 5.3 pede que se acrescente o estado Owned ao protocolo snooping MSI simples. Repita a questão, mas com o protocolo de diretório simples anterior.

5.12 [25] <5.4> Discuta por que acrescentar o estado Exclusivo é algo muito mais difícil de fazer com um protocolo de diretório simples do que com um protocolo snooping. Dê um exemplo dos tipos de problema que aparecem.

Estudo de caso 3: Protocolo avançado de diretório

Conceitos ilustrados por este estudo de caso

- Implementação do protocolo de coerência de diretório
- Desempenho do protocolo de coerência
- Otimizações do protocolo de coerência

O protocolo de coerência de diretório no Estudo de Caso 2 descreve a coerência de diretório no nível abstrato, mas assume que as transações são indivisíveis, de modo semelhante ao sistema de snooping simples. Os sistemas de diretório de alto desempenho utilizam interconexões de pipeline, comutadas, que melhoram bastante a largura de banda, mas também introduzem estados transientes e transações não indivisíveis (ou não atômicas). Os protocolos de coerência de cache de diretório são mais escaláveis do que os protocolos de coerência de cache snooping por dois motivos: 1) os protocolos de coerência de cache snooping enviam solicitações por broadcast a todos os nós, limitando sua escalada. Os protocolos de diretório utilizam um nível de indireção — uma mensagem ao diretório — para garantir que as solicitações só sejam enviadas aos nós que possuem cópias de um bloco; 2) o endereço de rede de um sistema snooping precisa entregar solicitações em uma ordem total, enquanto os protocolos de diretório podem relaxar essa restrição. Alguns protocolos de diretório não assumem qualquer ordenação de rede, o que é benéfico, pois isso permite que técnicas de roteamento adaptativas melhorem a largura de banda da rede. Outros protocolos contam com a ordem ponto a ponto (ou seja, as mensagens do nó P0 para o nó P1 chegarão em ordem). Mesmo com essa restrição de ordenação, os protocolos de diretório normalmente possuem mais estados transientes do que os protocolos snooping. A Figura 5.39 apresenta as

Estado	Leitura	Escrita	Substituição	INV	Forwarded_ GetS	Forwarded_ GetM	PutM_ Ack	Dados	Último ACK
I	Send GetS/IS^{D}	Send GetM/IM^{AD}	erro	Send Ack/I	erro	erro	erro	erro	erro
S	Faz Read	Send GetM/IM^{AD}	I	Send Ack/I	erro	erro	erro	erro	erro
M	Faz Read	Faz Write	Send PutM/MI^{A}	erro	Send Dados, Send PutMS/MS^{A}	Send Dados/I	erro	erro	erro
IS^{D}	z	z	z	Send Ack/ISI^{D}	erro	erro	erro	Salva Dados, faz Read/S	erro
ISI^{D}	z	z	z	Send Ack	erro	erro	erro	Salva Dados, faz Read/I	erro
IM^{AD}	z	z	z	Send Ack	erro	erro	erro	Salva Dados/IM^{A}	erro
IM^{A}	z	z	z	erro	IMS^{A}	IMI^{A}	erro	erro	Write/M
IMI^{A}	z	z	z	erro	erro	erro	erro	erro	Faz Write Send Dados/I
IMS^{A}	z	z	z	Send Ack/IMI^{A}	z	z	erro	erro	Faz Write, Send Dados/S
MS^{A}	Faz Read	z	z	erro	Send Dados	Send Dados MI^{A}	/S	erro	erro
MI^{A}	z	z	z	erro	Send Dados	Send Dados/I	/I	erro	erro

FIGURA 5.39 Transições de controlador de cache de snooping com broadcast.

transições de estado da unidade de controle de cache para um protocolo de diretório simplificado, que conta com uma ordenação de rede ponto a ponto. A Figura 5.40 apresenta as transições de estado da unidade de controle de diretório.

Estado	GetS	GetM	PutM (owner)	PutMS (não-owner)	PutM (owner)	PutMS (não-owner)
DI	Envia Data, adiciona a sharers/DS	Envia Data, apaga sharers, marca owner/DM	erro	Envia PutM_Ack	erro	Envia PutM_Ack
DS	Envia Data, adiciona a sharers/DS	Envia INVs a sharers, apaga sharers, marca owner, envia Data/DM	erro	Envia PutM_Ack	erro	Envia PutM_Ack
DM	Encaminha GetS, adiciona a sharers/ DMS^D	Encaminha GetM, envia INVs a sharers, apaga sharers, marca owner	Save Data, envia PutM_Ack/DI	Envia PutM_Ack	Save Data, adiciona a sharers, envia PutM_Ack/DS	Envia PutM_Ack
DMS^D	Encaminha GetS, adiciona a sharers	Encaminha GetM, envia INVs a sharers, apaga sharers, marca owner/DM	Save Data, envia PutM_Ack/DS	Envia PutM_Ack	Save Data, adiciona a sharers, envia PutM_Ack/DS	Envia PutM_Ack

FIGURA 5.40 Transições de controladora de diretório.

Para cada bloco, o diretório mantém um estado e um campo de proprietário atual ou uma lista de sharers atual (se houver). Para fins da discussão e consequentemente para o problema, considere que as caches L2 estão desabilitadas. Considere também que o diretório de memória lista sharers/owners em um processador de granularidade. Por exemplo, na Figura 5.38, o diretório para a linha 108 seria "`P0,0; P3,0`" em vez de "`C0,C1`". Além disso, suponha as mensagens através dos limites do chip — se necessários — de modo transparente.

As linhas, indexadas pelo estado atual, e as colunas, pelo evento, determinam a dupla <*ação/próximo estado*> . Se aparecer apenas um próximo estado, nenhuma ação será necessária. Casos impossíveis são marcados com "erro" e representam condições de erro; "z" significa que o evento solicitado não pode ser processado no momento.

O exemplo a seguir ilustra a operação básica desse protocolo. Suponha que um processador tente escrever em um bloco no estado I (Inválido). A tupla correspondente é "send GetM/IM^{AD}", indicando que a controladora de cache deverá enviar uma solicitação GetM (GetModified) ao diretório e passar para o estado IM^{AD}. No caso mais simples, a mensagem de solicitação encontra o diretório no estado DI (Directory Invalid), indicando que nenhuma outra cache possui uma cópia. O diretório responde com uma mensagem Data, que também contém o número de confirmações a esperar (nesse caso, zero). Nesse protocolo simplificado, a controladora da cache trata essa única mensagem como duas mensagens: uma mensagem Data, seguida por um evento Last Ack. A mensagem Data é processada primeiro, salvando os dados e passando para IM^A. O evento Last Ack é então processado, fazendo a transição para o estado M. Finalmente, a escrita pode ser realizada no estado M.

Se GetM encontrar o diretório no estado DS (Directory Shared), o diretório enviará mensagens de invalidação (INV) a todos os nós na lista de sharers, enviará Dados para o solicitante com o número de sharers e fará a transição para o estado M. Quando as mensagens INV chegarem aos sharers, encontrarão o bloco no estado S ou no estado I (se tiverem invalidado o bloco silenciosamente). De qualquer forma, o sharer enviará um Ack diretamente ao nó solicitante. O solicitante contará os Acks

que recebeu e comparará isso com o número enviado de volta com a mensagem Data. Quando todos os Acks tiverem chegado, ocorrerá o evento Last Ack, disparando a cache para fazer a transição para o estado M e permitindo que a escrita prossiga. Observe que é possível que todos os Acks cheguem antes da mensagem de Dados, mas não para que o evento Last Ack ocorra. Isso ocorre porque a mensagem de Dados contém o número de Ack. Assim, o protocolo considera que a mensagem de Dados é processada antes do evento Last Ack.

5.13 [10/10/10/10/10/10] <5.4> Considere o protocolo de diretório avançado que acabamos de descrever e o conteúdo de cache da Figura 5.38. Qual é a sequência de estados transientes que os blocos de cache afetados movimentam em cada um dos seguintes casos?

```
a. [10] <5.4> P0,0: read 100
b. [10] <5.4> P0,0: read 120
c. [10] <5.4> P0,0: write 120 <-- 80
d. [10] <5.4> P3,1: write 120 <-- 80
e. [10] <5.4> P1,0: read 110
f. [10] <5.4> P0,0: write 108 <-- 48
```

5.14 [15/15/15/15/15/15/15] <5.4> Considere o protocolo de diretório avançado descrito anteriormente e o conteúdo de cache da Figura 5.38. Qual é a sequência de estados transientes que os blocos de cache afetados movimentam em cada um dos seguintes casos? Em todos os casos, considere que os processadores emitem suas solicitações no mesmo ciclo, mas o diretório pede as solicitações na ordem de cima para baixo. Considere que as ações das controladoras parecem ser indivisíveis (p. ex., a controladora de diretório realizará todas as ações necessárias para a transição DS $\rightarrow$ DM antes de tratar de outra solicitação para o mesmo bloco).

```
a. [15] <5.4> P0,0: read 120
              P1,0: read 120
b. [15] <5.4> P0,0: read 120
              P1,0: write 120 <-- 80
c. [15] <5.4> P0,0: write 120
              P1,0: read 120
d. [15] <5.4> P0,0: write 120 <-- 80
              P1,0: write 120 <-- 90
e. [15] <5.4> P0,0: replace 110
              P1,0: read 110
f. [15] <5.4> P1,0: write 110 <-- 80
              P0,0: replace 110
g. [15] <5.4> P1,0: read 110
              P0,0: replace 110
```

5.15 [20/20/20/20/20] <5.4> Para o multiprocessador ilustrado na Figura 5.37 (com as caches L2 desabilitadas), implementando o protocolo descrito nas Figuras 5.39 e 5.40, considere as seguintes latências:

- Acertos de leitura e escrita da CPU não geram ciclos de stall.
- Para completar uma falta (ou seja, faz Read e faz Write), serão necessários L_{ack} ciclos *somente* se ela for realizada em resposta ao evento Last Ack (caso contrário, ela é feita enquanto os dados são copiados para a cache).
- Uma leitura ou escrita da CPU que gere um evento de substituição emite a mensagem GetShared ou GetModified correspondente antes da mensagem PutModified (p. ex., usando o buffer write-back).
- Um evento da unidade de controle de cache que envia uma mensagem de solicitação ou confirmação (p. ex., GetShared) tem latência de L_{send_msg} ciclos.
- Um evento da unidade de controle de cache que lê a cache e envia uma mensagem de dados tem latência L_{send_data} ciclos.

- Um evento da unidade de controle de cache que recebe uma mensagem de dados e atualiza a cache tem latência L_{rcv_data}.
- Uma unidade de controle de memória tem uma latência L_{send_msg} quando encaminha uma mensagem de solicitação.
- Uma unidade de controle de memória gera L_{inv} ciclos adicionais para cada invalidação que precisa enviar.
- Uma unidade de controle de cache tem latência L_{send_msg} para cada invalidação que recebe (a latência é até que ela envie a mensagem Ack).
- Uma unidade de controle de memória tem latência de L_{read_memory} ciclos para ler a memória e enviar uma mensagem de dados.
- Uma unidade de controle de memória tem latência L_{write_memory} para escrever uma mensagem de dados para a memória (a latência é até que ela envie a mensagem Ack).
- Uma mensagem não de dados (p. ex., solicitação, invalidação, Ack) tem latência de rede de L_{req_msg} ciclos.
- Uma mensagem de dados tem latência de rede de L_{data_msg} ciclos.
- Adicione uma latência de 20 ciclos para qualquer mensagem que atravesse do chip 0 para o chip 1 e vice-versa.

Considere uma implementação com as características de desempenho resumidas na Figura 5.41.

Ação	Latência
Send_msg	6
Send_data	20
Rcv_data	15
Read-memory	100
Write-memory	20
inv	1
ack	4
Req-msg	15
Data-msg	30

FIGURA 5.41 Latências de coerência de diretório.

Para as sequências de operações a seguir, o conteúdo da cache da Figura 5.38 e o protocolo de diretório apresentado, qual é a latência observada por nó de processador?

```
a. [20] <5.4> P0,0: read 100
b. [20] <5.4> P0,0: read 128
c. [20] <5.4> P0,0: write 128 <-- 68
d. [20] <5.4> P0,0: write 120 <-- 50
e. [20] <5.4> P0,0: write 108 <-- 80
```

5.16 [20] <5.4> No caso de uma falta de cache, tanto o protocolo switched snooping, descrito anteriormente, quanto o protocolo de diretório neste estudo de caso realizam a operação de leitura ou escrita assim que possível. Em particular, eles realizam a operação como parte da transição para o estado estável, em vez de fazer a transição para o estado estável e simplesmente tentar a operação novamente.

Isso *não* é uma otimização. Mais precisamente, para garantir o progresso do encaminhamento, as implementações de protocolo precisam garantir que realizam pelo menos uma operação de CPU antes de abandonar um bloco. Suponha que a implementação do protocolo de coerência não tenha feito isso. Explique como isso poderia levar ao livelock. Dê um exemplo de código simples que poderia estimular esse comportamento.

5.17 [20/30] <5.4> Alguns protocolos de diretório acrescentam o estado Owned (O) ao protocolo, de modo semelhante à otimização discutida para os protocolos snooping. O estado Owned comporta-se como o estado Shared, pois os nós só podem ler blocos Owned. Mas ele se comporta como o estado Modificado, no sentido de que os nós precisam fornecer dados nas solicitações Get de outros nós para os blocos Owned. O estado Owned elimina o caso em que uma solicitação GetShared a um bloco no estado Modificado exiga que o nó envie os dados tanto para o processador solicitante quanto para a memória. Em um protocolo de diretório MOSI, uma solicitação GetShared a um bloco nos estado Modificado ou Owned fornece dados para o nó solicitante e faz a transição para o estado Owned. Uma solicitação GetModified no estado Owned é tratada como uma solicitação no estado Modificado. Esse protocolo MOSI otimizado só atualiza a memória quando um nó substitui um bloco no estado Modificado ou Owned.

a. [20] <5.4> Explique por que o estado MSA no protocolo é basicamente um estado Owned "transiente".

b. [30] <5.4> Modifique as tabelas da cache e do protocolo de diretório para dar suporte a um estado Owned estável.

5.18 [25/25] <5.4> O protocolo avançado de diretório, descrito anteriormente, conta com uma interconexão ordenada ponto a ponto para garantir a operação correta. Considerando o conteúdo inicial da cache da Figura 5.38 e as sequências de operações a seguir, explique que problema poderia surgir se a interconexão deixasse de manter a ordenação ponto a ponto. Considere que os processadores realizam as solicitações ao mesmo tempo, mas são processadas pelo diretório na ordem mostrada.

```
a. [25] <5.4>     P1,0: read 110
                  P3,1: write 110 <--90
b. [25] <5.4>     P1,0: read 110
                  P0,0: replace 110
```

Exercícios

5.19 [15] <5.1> Considere que temos uma função para uma aplicação na forma $F(i, p)$, que dá a fração de tempo em que exatamente i processadores são utilizáveis, dado que um total de p processadores está disponível. Isso significa que:

$$\sum_{i=1}^{p} F(i,p) = 1$$

Considere que, quando i processadores estão em uso, as aplicações rodam i vezes mais rápido. Reescreva a lei de Amdahl de modo que ela forneça o ganho de velocidade como função de p para alguma aplicação.

5.20 [15/20/10] <5.1> Neste exercício, nós examinamos o efeito da topologia da rede de interconexão sobre os *ciclos de clock por instrução (CPI)* dos programas sendo executados em um multiprocessador de memória distribuída com 64 processadores. A frequência do processador é de 3,3 GHz e o CPI base de uma aplicação com todas as referências atingindo na cache é de 0,5. Considere que 0,2% das instruções envolvem uma referência de comunicação remota.

O custo de uma referência de comunicação remota é $(100 + 10h)$ *ns*, onde h é o número de caminhos na rede de comunicação que uma referência remota deve percorrer para a memória remota do processador e para a volta. Considere que todos os links de comunicação são bidirecionais.

a. [15] <5.1> Calcule o pior custo de comunicação remota quando os 64 processadores estão organizados como um anel, como um grid de 8:8 processadores ou como um hipercubo. (*Dica*: O maior caminho de comunicação em um hipercubo de 2^n tem n links).

b. [20] <5.1> Compare o CPI base da aplicação sem comunicação remota ao CPI alcançado com cada uma das três topologias no item *a*.

c. [10] <5.1> Quão mais rápida é a aplicação sem comunicação remota em comparação ao seu desempenho com a comunicação remota em cada uma das três topologias no item *a*?

5.21 [15] <5.2> Mostre como o protocolo básico snooping da Figura 5.7 pode ser modificado para uma cache write-through. Qual é a maior funcionalidade de hardware que não é necessária com uma cache write-through comparada com uma cache write-back?

5.22 [20] <5.2> Adicione um estado exclusivo limpo ao protocolo básico de coerência de cache snooping (Fig. 5.7). Mostre o protocolo no formato da Figura 5.7.

5.23 [15] <5.2> Uma solução proposta para o problema de falso compartilhamento é adicionar um bit de validade por palavra. Isso permitiria ao protocolo invalidar uma palavra sem remover o bloco inteiro, possibilitando que um processador mantenha uma parte de um bloco na sua cache enquanto outro escreve em uma parte diferente do bloco. Que complicações adicionais serão introduzidas no protocolo básico de coerência de cache snooping (Fig. 5.7) se esse recurso for incluído? Lembre-se de considerar todas as ações de protocolo possíveis.

5.24 [15/20] <5.3> Este exercício estuda o impacto de técnicas agressivas para explorar paralelismo em nível de instrução no processador quando usado no projeto de sistemas multiprocessadores com memória compartilhada. Considere dois sistemas idênticos, exceto pelo processador. O sistema *A* usa um processador com um pipeline simples de despacho único e em ordem, enquanto o sistema *B* usa um processador com despacho de quatro vias, execução fora de ordem e um buffer de reordenação com 64 entradas.

a. [15] <5.3> Seguindo a convenção na Figura 5.11, vamos dividir o tempo de execução em execução de instrução, acesso à cache, acesso à memória e outros stalls. Como você espera que cada um desses componentes seja diferente entre os sistemas *A* e *B*?

b. [10] <5.3> Baseado na discussão sobre o comportamento da carga de trabalho On-Line Transaction Processing (OLTP), na Seção 5.3, qual é a importante diferença entre a carga de trabalho OLTP e outros benchmarks que limitam o benefício de um projeto de processador mais agressivo?

5.25 [15] <5.3> Como você mudaria o código de uma aplicação para evitar compartilhamento falso? O que poderia ser feito por um compilador e o que requereria diretivas de programador?

5.26 [15] <5.4> Considere um protocolo de coerência de cache baseado em diretório. O diretório atualmente tem informações que indicam que o processador P1 tem os dados em modo "exclusivo". Se o diretório obtiver uma requisição para o mesmo bloco de cache do processador P1, o que isso poderia significar? O que o controlador de diretório faria? (Tais casos são chamados *condições de corrida* e são a razão pela qual os protocolos de coerência são tão difíceis de projetar e verificar.)

5.27 [20] <5.4> Um controlador de diretório pode enviar invalidações para linhas que tenham sido substituídas pelo controlador de cache local. Para evitar tais mensagens e manter o diretório consistente, são usadas dicas de substituição. Tais mensagens dizem ao controlador que um bloco foi substituído. Modifique o protocolo de coerência de diretório da Seção 5.4 para usar tais dicas de substituição.

5.28 [20/30] <5.4> Um ponto negativo de uma implementação direta de diretórios usando vetores de bit totalmente preenchidos é que o tamanho total da informação de diretório escala de acordo com o produto (p. ex., número de processadores $\times$ blocos de memórias). Se a memória crescer linearmente com o número de processadores, o tamanho total do diretório aumentará quadraticamente no número de processadores. Na prática, como o diretório precisa de somente um 1 bit por bloco de memória (geralmente, de 32-128 bytes), esse problema não é sério para números de processadores pequenos a moderados. Por exemplo, considerando um bloco de 128 bytes, a quantidade de armazenamento de diretório comparado à memória principal é o número de processadores/1024, ou cerca de 10% de armazenamento adicional com 100 processadores. Esse problema pode ser evitado observando que precisamos somente manter uma quantidade de informação proporcional ao tamanho da cache de cada processador. Nós exploramos algumas soluções nestes exercícios.

a. [20] <5.4> Um método para obter um protocolo de diretório escalável é organizar o multiprocessador como hierarquia lógica com os processadores como folhas da hierarquia e diretórios posicionados na raiz de tal subárvore. O diretório de cada subárvore registra que descendentes armazenam que blocos de memória na cache, além de quais blocos de memória com um home nessa subárvore são armazenados na cache fora da subárvore. Calcule a quantidade de armazenamento necessária para registrar as informações do processador para os diretórios, supondo que cada diretório seja totalmente associativo. Sua resposta deve também incorporar o número de nós em cada nível da hierarquia, além do número total de nós.

b. [30] <5.4> Uma técnica alternativa para implementar esquemas de diretório é implementar vetores de bits que não sejam densos. Existem duas estratégias: uma é reduzir o número de vetores de bit necessários e a outra é reduzir o número de bits por vetor. Usando traces, você pode comparar esses esquemas. Primeiro, implemente o diretório como uma cache associativa por conjunto de quatro vias armazenando vetores completos de bit, mas somente para blocos que são armazenados na cache fora do nó home. Se ocorrer falta de cache de diretório, selecione uma entrada de diretório e invalide a entrada. Segundo, implemente o diretório de modo que cada entrada tenha 8 bits. Se um bloco for armazenado na cache em somente um nó fora do seu home, esse campo conterá o número do nó. Se o bloco for armazenado na cache em mais de um nó fora do seu home, esse campo será um vetor de bit, com cada bit indicando um grupo de oito processadores, pelo menos um dos quais armazenará o bloco na cache. Usando trace de execução de 64 processadores, simule o comportamento desses esquemas. Considere uma cache perfeita para referências não compartilhadas, de modo a se concentrar em comportamento de coerência. Determine o número de invalidações externas conforme o aumento do tamanho da cache do diretório.

5.29 [10] <5.5> Implemente a instrução clássica test-and-set usando o par de instruções *load-linked/store-conditional*.

5.30 [15] <5.5> Uma otimização de desempenho comumente usada é preencher as variáveis de sincronização para que não tenham outros dados úteis na mesma linha de cache da variável de sincronização. Construir um exemplo patológico quando não se está fazendo isso pode prejudicar o desempenho. Considere um protocolo snooping de invalidação de escrita.

5.31 [30] <5.5> Uma implementação possível do par *load-linked/store-conditional* para processadores multicore é restringir o uso de operações de memória fora da cache a essas instruções. Uma unidade de monitoramento intercepta todas as leituras e gravações de qualquer núcleo para a memória. Ele rastreia a fonte das instruções *load-linked* e se quaisquer armazenamentos interligados ocorrem entre o *load-linked* e sua instrução *store-conditional* correspondente. O monitor pode impedir qualquer falha no armazenamento condicional de gravar qualquer dado e pode usar os sinais de interconexão para informar ao processador que esse armazenamento falhou. Projete um monitor desses para um sistema de memória suportando um multiprocessador simétrico de quatro núcleos (SMP).
Leve em conta que solicitações de leitura e gravação podem ter tamanhos diferentes de dados (4, 8, 16, 32 bytes). Qualquer local de memória pode ser o alvo de um par *load-linked/store-conditional*, e o monitor de memória deve supor que as referências de *load-linked/store-conditional* a qualquer local podem ser interligadas com acessos regulares ao mesmo local. A complexidade do monitor deve ser independente do tamanho da memória.

5.32 [10/12/10/12] <5.6> Como discutido na Seção 5.6, o modelo de consistência de memória fornece uma especificação de como o sistema de memória vai aparecer para o programador. Considere o seguinte segmento de código, onde os valores iniciais são

```
A=flag=C=0.
P1                      P2
A= 2000                 while (flag = =1){;}
flag=1                  C=A
```

a. [10] <5.6> No final do segmento de código, qual valor você esperaria para *C*?

b. [12] <5.6> Um sistema com uma rede de interconexão de uso geral, um protocolo de coerência de cache baseado em diretório e suporte para carregamentos sem bloqueio gera um resultado onde *C* é 0. Descreva um cenário onde esse resultado seja possível.

c. [10] <5.6> Se você quisesse tornar o sistema sequencialmente consistente, que restrições principais precisaria impor?
Considere que um processador suporte um modelo relaxado de consistência de memória. Esse modelo requer que a sincronização seja identificada explicitamente. Considere que o processador suporte uma instrução de "barrier", o que garante que todas as operações de memória precedendo essa instrução sejam completadas antes que quaisquer operações de memória após a barreira sejam autorizadas a começar. Onde você incluiria as instruções barrier no segmento de código anterior para garantir que obteria os "resultados intuitivos" da consistência sequencial?

5.33 [25] <5.7> Prove que, em uma hierarquia de cache de dois níveis, onde L1 está mais próxima ao processador, a inclusão é mantida sem ação adicional se L2 tiver pelo menos a mesma associatividade que L1, as duas caches usarem substituição por unidade substituível de linha (LRU) e as duas caches tiverem os mesmos tamanhos de bloco.

5.34 [Discussão] <5.7> Quando estão tentando realizar uma avaliação detalhada do desempenho de um sistema multiprocessador, os projetistas de sistema usam uma de três ferramentas: modelos analíticos, simulação orientada a trace e simulação orientada a execução. Modelos analíticos usam expressões matemáticas para modelar o comportamento dos programas. Simulações orientadas a trace rodam as aplicações em uma máquina real e geram o trace, em geral de operações de memória. Esses traces podem ser reproduzidos através de um simulador de cache ou um simulador com um modelo simples de processador para prever o desempenho do sistema quando vários parâmetros são modificados. Simuladores orientados a execução simulam toda a execução mantendo uma estrutura equivalente para o estado do processador, e assim por diante. Quais são as trocas entre precisão e velocidade entre essas abordagens?

5.35 [40] <5.7, 5.9> Multiprocessadores e clusters geralmente mostram aumentos de desempenho conforme você aumenta o número de processadores, com o ideal sendo n $\times$ o ganho de velocidade para *n* processadores. O objetivo desse benchmark enviesado é fazer com que um programa tenha um desempenho pior conforme a adição de processadores. Isso quer dizer, por exemplo, que um processador no multiprocessador ou cluster executa o programa mais rapidamente, dois são mais lentos, quatro são mais lentos que dois, e assim por diante. Quais são as principais características de desempenho para cada organização que gera ganho de velocidade linear inverso?

CAPÍTULO 6

Computadores em escala warehouse para explorar paralelismo em nível de requisição e em nível de dados

O datacenter é o computador.

Luiz André Barroso, *Google* (2007)

Cem anos atrás, as empresas pararam de gerar sua própria energia
com motores a vapor e dínamos e se ligaram à recém-instalada rede elétrica.
A energia barata fornecida pelos serviços de eletricidade não mudou apenas
o modo como as empresas operavam. Ela iniciou uma reação em cadeia
de transformações econômicas e sociais que deu à luz o mundo moderno. Hoje,
uma revolução similar está em andamento. Ligadas à rede de computação global
da internet, enormes fábricas de processamento de informações começaram
a enviar dados e códigos de software para nossas casas e empresas. Dessa vez,
é a computação que está se transformando em um serviço.

Nicholas Carr, *The Big Switch: Rewiring the World, from Edison to Google (2008)*

6.1 INTRODUÇÃO

Qualquer um pode construir uma CPU rápida. O truque é construir um sistema rápido.

Seymour Cray, *Considerado o Pai dos Supercomputadores*

O computador em escala warehouse (Warehouse-Scale Computer— WSC)[i] é a base dos serviços da internet que muitas pessoas usam todos os dias: busca, redes sociais, mapas

[i] Este capítulo baseia-se em conteúdos do livro *The Datacenter as a Computer: An Introduction to the Design of Warehouse-Scale Machines*, de Luiz André Barroso e Urs Hölzle, do Google (2009); o blog *Perspectives* em mvdirona.com e as palestras "Cloud-Computing Economies of Scale" e "Data Center Networks Are in My Way", de James Hamilton, da Amazon Web Services (2009, 2010); e o relatório técnico *Above the Clouds: A Berkeley View of Cloud Computing*, de Michael Armbrust *et al.* (2009).

on-line, compartilhamento de vídeo, compras on-line, serviços de e-mail, e assim por diante. A enorme popularidade de tais serviços da internet exigiram a criação de WSCs que pudessem acompanhar as rápidas demandas do público. Embora os WSCs possam apenas parecer grandes datacenters, sua arquitetura e sua operação são muito diferentes, como veremos. Os WSCs de hoje agem como uma grande máquina e custam cerca de US$ 150 milhões, incluindo as instalações, as infraestruturas elétrica e de refrigeração, os servidores e o equipamento de rede que conecta e mantém 50.000-100.000 servidores. Além do mais, o rápido crescimento da computação em nuvem (Seção 6.5) torna os WSCs disponíveis para qualquer pessoa que tenha um cartão de crédito.

A arquitetura de computadores se estende naturalmente ao projeto de WSCs. Por exemplo, Luiz Barroso, do Google (citado anteriormente), fez sua pesquisa de dissertação sobre arquitetura de computadores. Ele acredita que as habilidades de um arquiteto de projetar para escala, projetar para confiabilidade e dar um "jeito" para depurar hardware são muito úteis na criação e na operação de WSCs.

Nessa escala extrema, que requer inovação na distribuição de energia, monitoramento da refrigeração e operações, o WSC é o descendente moderno do supercomputador, fazendo de Seymour Cray o padrinho dos arquitetos de WSC atuais. Seus computadores extremos lidavam com cálculos que não poderiam ser feitos em nenhum outro lugar, mas eram tão caros que poucas empresas podiam pagar por eles. Dessa vez, o objetivo é fornecer ao mundo tecnologia da informação em vez de computação de alto desempenho (High-Performance Computing — HPC) para cientistas e engenheiros. Portanto, pode-se argumentar que os WSCs têm um papel mais importante para a sociedade de hoje do que os supercomputadores de Cray tinham no passado.

Sem dúvida, os WSCs têm um número de usuários muito maior do que a computação de alto nível e representam uma parcela muito maior do mercado de TI. Seja medido pelo número de usuários, seja medido pela receita, o Google é pelo menos 250 vezes maior do que a Cray Research.

Os arquitetos de WSC têm muitos objetivos e demandas em comum com os arquitetos de servidor:

- *Custo-desempenho.* O trabalho realizado por dólar é crítico, em parte por causa da escala. Reduzir o custo capital de um WSC em 10% poderia poupar US$ 15 milhões.
- *Eficiência energética.* Os custos da distribuição de energia estão funcionalmente relacionados ao consumo de energia. Você precisa de distribuição de energia suficiente antes de consumi-la. Os custos do sistema mecânico estão funcionalmente relacionados à energia: você precisa colocar para fora o calor que coloca para dentro. Portanto, o pico de energia e a energia consumida orientam o custo da distribuição de energia e o custo dos sistemas de refrigeração. Além do mais, a eficiência energética é uma parte importante da administração ambiental. Portanto, o trabalho realizado por joule é fundamental para os WSCs e para os servidores, devido ao alto custo de construção de infraestruturas energética e mecânica para um depósito de computadores e para as contas mensais de eletricidade usada para alimentar os servidores.
- *Confiabilidade através de redundância.* A natureza de longo prazo dos serviços de internet significa que o hardware e o software, em um WSC, devem fornecer juntos pelo menos 99,99% de disponibilidade. Ou seja, eles devem estar indisponíveis menos de uma hora por ano. A redundância é a chave para a confiabilidade para os WSCs e para os servidores. Enquanto, muitas vezes, os arquitetos de servidores utilizam mais hardware oferecido a custos mais altos para atingir alta

disponibilidade, os arquitetos de WSC dependem de múltiplos servidores efetivos em termos de custos, conectados por uma rede de baixo custo e uma redundância administrada por software. Além do mais, se o objetivo é ir muito além de "quatro noves" de disponibilidade, você precisa de múltiplos WSCs para mascarar eventos que podem derrubar WSCs inteiros. Múltiplos WSCs também reduzem a latência para serviços que são muito empregados.

- *E/S de rede.* Os arquitetos de servidor devem fornecer uma boa interface de rede para o mundo externo, assim como os arquitetos de WSC. A rede é necessária para manter os dados consistentes entre múltiplos WSCs, além de fazer a interface com o público.
- *Cargas de trabalho interativas e de lote.* Embora você espere cargas de trabalho altamente interativas para serviços como busca e redes sociais com milhões de usuários, os WSCs, assim como os servidores, também rodam programas em lote altamente paralelos para calcular metadados úteis para esses serviços. Por exemplo, os serviços MapReduce são executados para converter as páginas retornadas do "crawling" na Web em índices de busca (Seção 6.2).

Não é de surpreender que também existam características *não* compartilhadas com a arquitetura de servidor:

- *Amplo paralelismo.* Uma preocupação, para um arquiteto de servidor, é se as aplicações no mercado-alvo têm paralelismo suficiente para justificar a quantidade de hardware paralelo e se o custo é muito alto para um hardware de comunicação suficiente para explorar esse paralelismo. Um arquiteto de WSC não tem tal preocupação: 1) as aplicações em lote se beneficiam do grande número de conjuntos de dados independentes que requerem processamento independente, como os bilhões de Web pages de um Web crawl. Esse processamento é *paralelismo em nível de dados* aplicado a dados em armazenamento em discos em vez de dados na memória, que vimos no Capítulo 4; 2) aplicações de serviço interativo na internet, também conhecidas como *software como serviço* (Software as a Service — SaaS), podem se beneficiar dos milhões de usuários independentes de serviços interativos da internet. Leitura e escrita quase nunca são dependentes em SaaS, por isso ele raramente precisa sincronizar. Por exemplo, buscas usam um índice somente leitura, normalmente, e e-mails são informações independentes de leitura e escrita. Chamamos esse tipo de paralelismo fácil de *paralelismo em nível de requisição*, já que muitos esforços independentes podem proceder naturalmente em paralelo, com pouca necessidade de comunicação ou sincronização. Por exemplo, a atualização jornal-based pode reduzir as demandas de throughput. Dado o sucesso do SaaS e dos WSCs, aplicações mais tradicionais, como bases de dados relacionais, foram enfraquecidas para depender do paralelismo em nível de requisição. Mesmo recursos dependentes de leitura/escrita às vezes são abandonados para oferecer armazenamento em discos que pode ser escalado para o tamanho dos modernos WSCs.
- *Estimativa dos custos operacionais.* Os arquitetos de servidor geralmente projetam seus sistemas para o pico de desempenho dentro de um orçamento de custos e se preocupam com a energia somente para garantir que eles não excedam a capacidade de resfriamento do seu invólucro. Eles geralmente ignoram os custos operacionais de um servidor, presumindo que são baixos em comparação aos custos de aquisição. Os WSCs têm tempos de vida mais longos — muitas vezes, o prédio e a infraestrutura elétrica e de refrigeração são amortizados ao longo de 10 anos ou mais —, então os custos operacionais se somam: energia, distribuição de energia e refrigeração representam mais de 30% dos custos de um WSC em 10 anos.

- *Escala e as oportunidades/problemas associados com a escala.* Muitas vezes, os computadores extremos são muito caros porque requerem hardware personalizado, mas o custo da personalização não pode ser amortizado efetivamente, uma vez que poucos computadores extremos são construídos. Entretanto, quando você compra 50.000 servidores e a infraestrutura que os acompanha para construir um único WSC, *consegue* descontos por volume. Os WSCs são tão grandes internamente que você pode obter economia de escala mesmo que não haja tantos WSCs. Como veremos nas Seções 6.5 e 6.10, essas economias de escala levam à computação em nuvem, já que os custos por unidades menores de um WSC significam que as empresas podem alugá-los com lucro, com preço abaixo do que custaria para pessoas de fora fazê-lo por si próprias. O outro lado de 50.000 servidores são as falhas. A Figura 6.1 mostra as indisponibilidade e anomalias de 2.400 servidores. Mesmo que um servidor tivesse um tempo médio para falha (Mean Time to Failure — MTTF) de incríveis 25 anos (200.000 horas), o arquiteto de WSC precisaria projetar para cinco falhas de servidor por dia. A Figura 6.1 lista a taxa anual de falha de disco como 2-10%. Se houvesse quatro discos por servidor e sua taxa de falhas anuais fosse de 4% com 50.000 servidores, o arquiteto de WSC deveria esperar ver uma falha de disco por *hora*.

Exemplo Calcule a disponibilidade de um serviço executando nos 2.400 servidores na Figura 6.1. Ao contrário de um serviço em um WSC real, neste exemplo os serviços não podem tolerar falhas de hardware ou software. Considere que o tempo para reiniciar o software seja de cinco minutos e o tempo para reparar o hardware seja de uma hora.

Resposta Podemos estimar a disponibilidade de serviço calculando o tempo de indisponibilidades devidas a falhas de cada componente. Vamos, conservadoramente, tomar o menor número de cada categoria na Figura 6.1 e dividir as 1.000 indisponibilidades igualmente entre os quatro componentes. Nós ignoramos discos lentos — o quinto componente das 1.000 indisponibilidades —, já que eles prejudicam o desempenho, mas não a disponibilidade e falhas na rede elétrica, já que o sistema de fornecimento ininterrupto de energia (Uninterrupted Power Supply — UPS) oculta 99% delas.

$$\text{Horas Indisponibilidade}_{\text{serviço}} = (4+250+250+250)\times 1\,\text{hora} + (250+5.000)\times 5\,\text{minutos}$$

$$= 754 + 438 = 1.192\,\text{horas}$$

Uma vez que existem 365 × 24 ou 8.760 horas em um ano, a disponibilidade é:

$$\text{Disponibilidade}_{\text{sistema}} = \frac{(8.760 - 1.192)}{8.760} = \frac{7.598}{8.760} = 86\%$$

Ou seja, sem redundância de software para mascarar as muitas indisponibilidades, um serviço nesses 2.400 servidores estaria fora do ar em média um dia por semana, ou *zero* noves de disponibilidade!

Conforme a Seção 6.10 explica, os precursores dos WSCs são *clusters de computadores*: clusters são coleções de computadores independentes conectados usando redes locais padrão (Local Area Networks — LANs) e switches de prateleira. Para cargas de trabalho que não precisavam de comunicação intensa, os clusters ofereciam computação muito mais efetiva em termos de custos do que os multiprocessadores de memória compartilhada. (Os multiprocessadores de memória compartilhada foram os precursores dos computadores multicore discutidos no Capítulo 5.) Os clusters se tornaram populares no final dos anos 1990 para computação científica e mais tarde para serviços de internet. Um modo de ver os WSCs é que eles são apenas a evolução lógica dos clusters de centenas de servidores às dezenas de milhares de servidores de hoje.

Número aprox. de eventos no 1º ano	Causa	Consequência
1 ou 2	Falhas da rede elétrica	Perda de energia em todo o WSC; não derruba o WSC se o UPS e os geradores funcionarem (ou geradores funcionam cerca de 99% do tempo).
4	Atualizações de cluster	Indisponibilidade planejada para atualizar a infraestrutura, muitas vezes para melhorias de rede, como recabeamento, atualizações de firmware, e assim por diante. Há cerca de 9 indisponibilidades de cluster planejadas para cada indisponibilidade não planejada.
1.000	Falhas de disco rígido	2-10% de taxa anual de falha de disco (Pinheiro, 2007).
	Discos lento	Ainda opera, mas roda 10-20× mais lentamente.
	Memórias ruins	Um erro incorrigível de DRAM por ano (Schroeder *et al*., 2009).
	Máquinas mal configuradas	A configuração leva a ~30% de interrupção do serviço (Barroso e Hölzle, 2009).
	Máquinas não confiáveis	1% dos servidores reiniciados mais de uma vez por semana (Barroso e Hölzle, 2009).
5.000	Crashes individuais de servidor	A máquina é reinicializada, geralmente leva cerca de 5 minutos.

FIGURA 6.1 Lista de indisponibilidades e anomalias com as frequências aproximadas de ocorrência no primeiro ano de um novo cluster de 2.400 servidores.
O que o Google chama cluster nós chamamos *array*. Veja a Figura 6.5. *(Baseada em Barroso, 2010.)*

Uma questão natural é se os WSCs são similares aos clusters modernos para computação de alto desempenho. Embora alguns tenham escala e custo similares — existem projetos HPC com um milhão de processadores que custam centenas de milhares de dólares —, geralmente têm processadores muito mais rápidos e redes muito mais rápidas entre os nós do que os encontrados nos WSCs, porque as aplicações HPC são mais interdependentes e se comunicam com mais frequência (Seção 6.3). Os projetos HPC também tendem a usar hardware customizados — especialmente na rede —, por isso muitas vezes não têm os benefícios do custo de usar chips encontrados no mercado. Por exemplo, só o microprocessador IBM Power7 pode custar mais e usa mais energia do que um nó de servidor em um WSC Google. O ambiente de programação também enfatiza o paralelismo em nível de thread ou paralelismo em nível de dados (Caps. 4 e 5), geralmente enfatizando a latência para completar uma única tarefa em vez da largura de banda para completar muitas tarefas independentes através do paralelismo em nível de requisição. Os clusters HPC também tendem a ter serviços de longa duração que mantêm os servidores totalmente utilizados, durante semanas, enquanto a utilização de servidores em WSCs varia de 10-50% (Fig. 6.3, na página 387), e varia todos os dias.

Como os WSCs se comparam a datacenters convencionais? Os operadores de um datacenter convencional geralmente coletam máquinas e software de terceiros de muitas partes de uma organização e os executam centralmente para outros. Seu foco principal tende a ser a consolidação dos muitos serviços em menos máquinas, que são isoladas umas das outras para proteger informações sensíveis. Portanto, as máquinas virtuais são cada vez mais importantes nos datacenters. Ao contrário dos WSCs, os datacenters convencionais tendem a ter muita heterogeneidade de hardware e software para servir seus vários clientes dentro de uma organização. Os programadores de WSC personalizam software de terceiros ou criam seus próprios, e os WSCs têm hardware muito mais homogêneo. O objetivo do WSC é levar o hardware/software no depósito a agir como um único computador que executa uma variedade de aplicações. Muitas vezes, o maior custo em um datacenter convencional são as pessoas que o mantêm, enquanto, como veremos na Seção 6.4, em um WSC bem projetado o hardware é o maior custo, e os custos com pessoal variam de altos a quase irrelevantes. Datacenters convencionais também não têm a escala de um WSC,

então não obtêm os benefícios econômicos de escala mencionados. Portanto, enquanto você pode considerar um WSC um datacenter extremo, no qual os computadores são alocados separadamente em um espaço com infraestrutura energética e de refrigeração especial, os datacenter típicos compartilham pouco dos desafios e oportunidades de um WSC, seja arquitetonicamente, seja operacionalmente.

Uma vez que poucos arquitetos entendem o software que roda em um WSC, nós começamos com a carga de trabalho e modelo de programação de um WSC.

6.2 MODELOS DE PROGRAMAÇÃO E CARGAS DE TRABALHO PARA COMPUTADORES EM ESCALA WAREHOUSE

> Se um problema não tem solução, pode não ser um problema, mas um fato — não para ser solucionado, mas com o qual se deve lidar ao longo do tempo.
>
> **Shimon Peres**

Além dos serviços públicos da internet, como busca, compartilhamento de vídeo e redes sociais, que a tornaram famosa, os WSCs também executam aplicações de lote, como converter vídeos em novos formatos ou criar índices de busca a partir de Web crawls.

Hoje, o framework mais popular para processamento de lote em um WSC é MapReduce (Dean e Ghemawat, 2008) e seu gêmeo open source Haddop. A Figura 6.2 mostra a crescente popularidade do MapReduce no Google ao longo do tempo. (O Facebook executou o Hadoop em 2.000 servidores de processadores de lote dos 60.000 servidores que se estima que ele tivesse em 2011.) Inspirado pelas funções Lisp de mesmo nome, o Map primeiro aplica uma função fornecida pelo programador para cada registro de entrada lógica. O Map é executado em milhares de computadores para produzir um resultado intermediário de pares de valores-chave. Reduz a coleta da saída dessas tarefas distribuídas e as compacta usando outra função definida pelo programador. Com suporte apropriado de software, os dois são altamente paralelos e, ainda, fáceis de entender e usar. Dentro de 30 minutos, um programador novato pode executar uma tarefa MapReduce em milhares de computadores.

Por exemplo, um programa MapReduce calcula o número de ocorrências de cada palavra da língua inglesa em um grande conjunto de documentos. A seguir apresentamos uma versão simplificada desse programa, que mostra somente o loop interno e supõe somente uma ocorrência de todas as palavras da língua inglesa em um documento (Dean e Ghemawat, 2008):

```
map(String key, String value):
      //key: nome do documento
      //value: conteúdo do documento
      para cada palavra w em value:
      EmitIntermediate(w, “1”); // Produz a lista de todas as palavras
reduce(String key, Iterator values):
      //key: uma palavra
      //values: uma lista de counts
      int result = 0;
      para cada v em values:
      result += ParseInt(v); // obtém inteiro do par key-value
      Emit(AsString(result));
```

	4 Ago	6 Mar	7 Set	9 Set
Número de jobs MapReduce	29.000	171.000	2.217.000	3.467.000
Tempo médio de conclusão (segundos)	634	874	395	475
Anos de uso de servidor	217	2.002	11.081	25.562
Dados de entrada lidos (terabytes)	3.288	52.254	403.152	544.130
Dados intermediários (terabytes)	758	6.743	34.774	90.120
Dados de saída escritos (terabytes)	193	2.970	14.018	57.520
Número médio de servidores por serviço	157	268	394	488

FIGURA 6.2 **Uso anual de MapReduce no Google ao longo do tempo.**
Ao longo de cinco anos, o número de jobs MapReduce aumentou por um fator de 100 e o número médio de servidores por job aumentou por um fator de 3. Nos últimos dois anos, os aumentos foram por fatores de 1,6 e 1,2, respectivamente (Dean, 2009). A Figura 6.16, na página 404, estima que executar a carga de trabalho de 2009 no serviço de computação em nuvem da Amazon, EC2 custaria US$ 133 milhões.

A função EmitIntermediate usada na função Map emite cada palavra no documento e o valor 1. Então, a função Reduce soma todos os valores por palavra para cada documento usando ParseInt() para obter o número de ocorrências por palavra em todos os documentos. O ambiente de execução do MapReduce escalona tarefas map e reduz para os nós de um WSC (a versão completa do programa é encontrada em Dean e Ghemawat, 2004).

Pode-se pensar no MapReduce como uma generalização da operação SIMD (Cap. 4) — exceto pelo fato de que você envia uma função a ser aplicada aos dados —, que é seguida por uma função usada em uma redução da saída da tarefa Map. Já que as reduções são comuns até mesmo em programas SIMD, muitas vezes o hardware SIMD oferece a elas operações especiais. Por exemplo, as recentes instruções AVX SIMD da Intel incluem instruções "horizontais" que somam pares de operandos adjacentes em registradores.

Para acomodar a variabilidade em desempenho de milhares de computadores, o escalonador do MapReduce designa novas tarefas com base na rapidez com que ele completa as tarefas anteriores. Obviamente, uma única tarefa lenta pode atrasar a conclusão de um grande serviço MapReduce. Em um WSC, a solução para tarefas lentas é fornecer mecanismos de software para lidar com a variabilidade inerente a essa escala. Tal abordagem está em contraste direto com a solução para um servidor em um datacenter convencional, em que tarefas tradicionalmente lentas significam que o hardware está quebrado e precisa ser substituído ou que o software do servidor precisa ser ajustado e reescrito. A heterogeneidade de desempenho é a norma para 50.000 servidores em um WSC. Por exemplo, próximo ao final de um programa MapReduce, o sistema vai iniciar execuções de backup em outros nós das tarefas que ainda não foram completadas e usar o resultado de qualquer um que acabe primeiro. Em troca de aumentar o uso de recursos alguns pontos porcentuais, Dean e Ghemawat (2008) descobriram que algumas tarefas grandes são completadas 30% mais rápido.

Outro exemplo de como os WSCs são diferentes é o uso de replicação de dados para superar as falhas. Dada a quantidade de equipamento em um WSC, não é de surpreender que as falhas sejam comuns, como o exemplo anterior atesta. Para fornecer 99,99% de disponibilidade, os sistemas de software devem lidar com essa realidade em um WSC. Para reduzir os custos operacionais, todos os WSCs usam softwares de monitoramento automático, de modo que um operador pode ser responsável por mais de 1.000 servidores.

O sucesso de frameworks de programação, como o MapReduce para processamento de lote e SaaS de uso público, como buscas, depende de serviços internos de software. Por exemplo, o MapReduce depende do Sistema de Arquivos Google (Google File System —

GFS) (Ghemawat, Gobioff e Leung, 2003) para fornecer arquivos a qualquer computador, de modo que as tarefas do MapReduce possam ser escalonadas em qualquer lugar.

Além do GFS, exemplos de tais sistemas de armazenamento escaláveis incluem o sistema de armazenamento de valores-chave da Amazon, Dynamo (DeCandia *et al.*, 2007), e o sistema de armazenamento de registros do Google, Bigtable (Chang, 2006). Note que tais sistemas muitas vezes se empilham uns sobre os outros. Por exemplo, o Bigtable armazena seus logs e dados no GFS, de modo similar à que uma base de dados relacional pode usar o sistema de arquivos fornecido pelo kernel do sistema operacional.

Muitas vezes, esses serviços internos tomam decisões diferentes das de softwares similares sendo executados em servidores únicos. Como exemplo, em vez de supor que o armazenamento é confiável, como ao usar servidores de armazenamento RAID, esses sistemas muitas vezes criam réplicas completas dos dados. Réplicas podem ajudar o desempenho da leitura, assim como a disponibilidade. Com o posicionamento correto, réplicas podem superar muitas outras falhas de sistema, como as da Figura 6.1. Alguns sistemas usam codificação por eliminação (do inglês *erasure encoding*), em vez de réplicas completas, mas a constante é a redundância através dos servidores em vez de redundância dentro de um servidor ou redundância em um array de armazenamento. Portanto, uma falha de todo o servidor ou dispositivo de armazenamento não afeta negativamente a disponibilidade dos dados.

Outro exemplo da técnica diferente é que, muitas vezes, o software de armazenamento dos WSCs usa consistência relaxada em vez de seguir todos os requisitos ACID (atomicidade, consistência, isolamento e durabilidade) dos sistemas de base de dados convencionais. A percepção importante é que múltiplas réplicas dos dados *eventualmente* estejam de acordo, mas, para a maior parte das aplicações, elas não precisam estar de acordo o tempo todo. Por exemplo, a consistência eventual é suficiente para compartilhamento de vídeo. A consistência eventual torna os sistemas de armazenamento muito mais fáceis de escalar, o que é um requerimento absoluto para os WSCs.

As demandas de carga de trabalho desses serviços interativos públicos variam consideravelmente. Até mesmo um serviço global popular como o Google pode variar por um fator de 2, dependendo da hora do dia. Quando você leva em conta fins de semana, feriados e épocas populares do ano para algumas aplicações — como serviços de compartilhamento de fotos depois do Halloween ou compras on-line antes do Natal —, pode ver uma variação consideravelmente maior na utilização de servidor para serviços de internet. A Figura 6.3 mostra a utilização média de 5.000 servidores do Google ao longo de um período de seis meses. Observe que menos de 0,5% dos servidores tiveram uma média de utilização de 100%, e a maioria dos servidores operou entre 10-50% de utilização. Em outras palavras, somente 10% dos servidores foram utilizados mais de 50%. Portanto, é muito mais importante para os servidores em um WSC ter bom desempenho fazendo pouco do que ter desempenho eficiente apenas em seu pico, já que raramente operam no pico.

Em resumo, o hardware e o software dos WSCs devem lidar com a variabilidade em carga com base na demanda do usuário, no desempenho e na confiabilidade, devido aos caprichos do hardware nessa escala.

Exemplo Como resultado de medições como as da Figura 6.3, o benchmark SPECPower mede o consumo de energia e o desempenho de carga de 0% a 100% em incrementos de 10% (Cap. 1). A única métrica geral que resume esse benchmark é a soma de todas as medidas de desempenho (operações Java por segundo do lado do servidor) dividida pela soma de todas as medições de potência em watts. Assim, cada nível é igualmente provável. Como os números resumem a mudança da métrica se os níveis forem ponderados pelas frequências de utilização na Figura 6.3?

Resposta A Figura 6.4 mostra os pesos originais e os novos pesos que correspondem à Figura 6.3. Esses pesos reduzem o resumo do desempenho em 30%, de 3.210 ssj_ops/watt para 2.454.

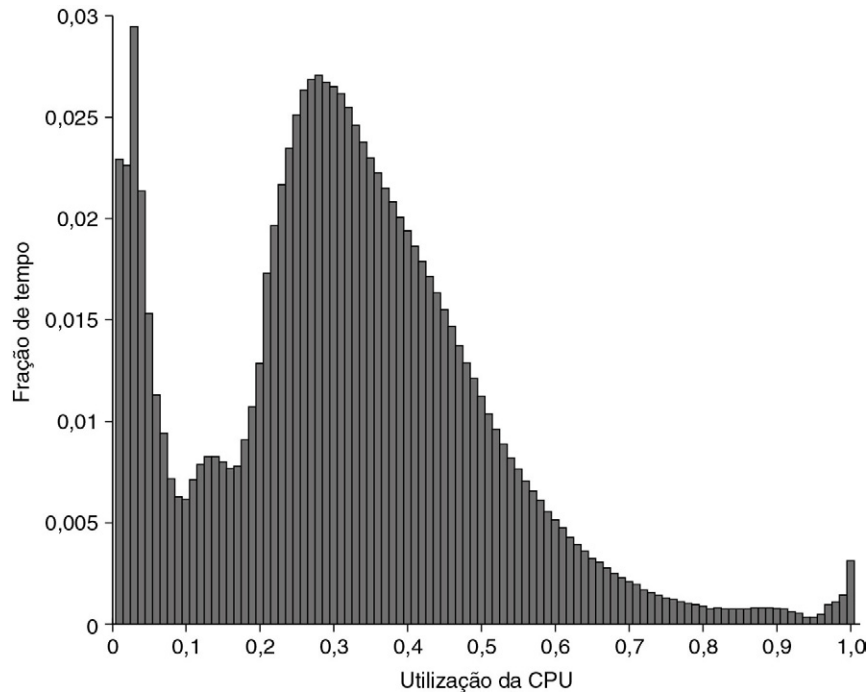



FIGURA 6.3 **Utilização média da CPU de mais de 5.000 servidores durante um período de seis meses no Google.**
Os servidores raramente estão completamente ociosos ou completamente utilizados, em vez disso operando a maioria do tempo entre 10-50% da sua utilização máxima. (Da figura 1 em Barroso e Hölzle, 2007.) A terceira coluna a partir da direita na Figura 6.4 calcula as porcentagens mais ou menos 5% para resultar nos pesos: assim, 1,2% para a linha de 90% significa que 1,2% dos servidores tinham utilização entre 85-95%.

Carga	Desempenho	Watts	Pesos SPEC	Desempenho ponderado	Watts ponderados	Pesos da Fig. 6.3	Desempenho ponderado	Watts ponderados
100%	2.889.020	662	9,09%	262.638	60	0,80%	22.206	5
90%	2.611.130	617	9,09%	237.375	56	1,20%	31.756	8
80%	2.319.900	576	9,09%	210.900	52	1,50%	35.889	9
70%	2.031.260	533	9,09%	184.660	48	2,10%	42.491	11
60%	1.740.980	490	9,09%	158.271	45	5,10%	88.082	25
50%	1.448.810	451	9,09%	131.710	41	11,50%	166.335	52
40%	1.159.760	416	9,09%	105.433	38	19,10%	221.165	79
30%	869.077	382	9,09%	79.007	35	24,60%	213.929	94
20%	581.126	351	9,09%	52.830	32	15,30%	88.769	54
10%	290.762	308	9,09%	26.433	28	8,00%	23.198	25
0%	0	181	9,09%	0	16	10,90%	0	20
Total	15.941.825	4.967		1.449.257	452		933.820	380
				ssj_ops/Watt	3.210		ssj_ops/Watt	2.454

FIGURA 6.4 **Resultado SPECPower da Figura 6.17 usando os pesos da Figura 6.3 em vez de pesos iguais.**

Dada a escala, o software deve lidar com falhas, o que significa que há pouca razão para comprar um hardware "revestido de ouro" que reduza a frequência de falhas. O principal impacto seria aumentar o custo. Barroso e Hölzle (2009) encontraram um fator de 20 para as diferenças em preço-desempenho entre um multiprocessador HP de memória compartilhada de alto nível e um servidor HP comercial ao executar o benchmark de base de dados TPC-C. Não é surpreendente que o Google compre servidores comerciais comuns.

Tais serviços WSC também tendem a desenvolver seu próprio software em vez de comprar softwares comerciais de terceiros, em parte para lidar com a grande escala e noutra para poupar dinheiro. Por exemplo, mesmo na melhor plataforma em termos de preço-desempenho para TPC-C em 2011, incluindo o custo da base de dados Oracle e o sistema operacional Windows, dobrava o custo do servidor Poweredge 710 da Dell. Em contraste, o Google executa o Bigtable e o sistema operacional Linux nos seus servidores, pelos quais ele não paga taxas de licenciamento.

Dados essa revisão das aplicações e o sistema de software de um WSC, estamos prontos para examinar a arquitetura de computador de um WSC.

6.3 ARQUITETURA DE COMPUTADORES EM ESCALA WAREHOUSE

As redes são o meio de conectividade que mantém 50.000 servidores juntos. De modo análogo à hierarquia de memória do Capítulo 2, os WSCs usam uma hierarquia de redes. A Figura 6.5 mostra um exemplo disso. De modo ideal, a rede combinada forneceria praticamente o desempenho de um switch personalizado de alto nível para 50.000 servidores

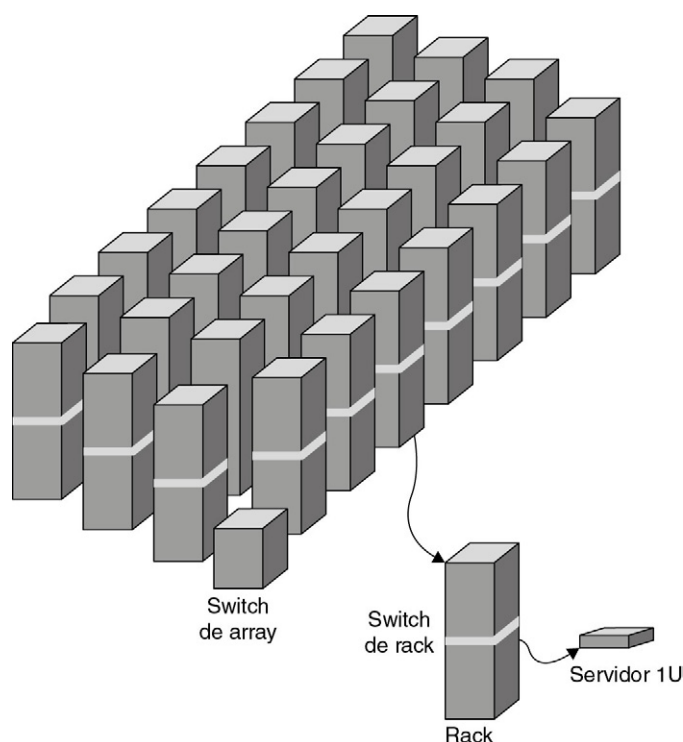



FIGURA 6.5 Hierarquia dos switches em um WSC.
(Baseado na Figura 1.2 de Barroso e Hölzlel, 2009.)

ao custo por porta de um switch comercial projetado para 50 servidores. Como veremos na Seção 6.6, as soluções atuais estão longe do ideal, e as redes para WSCs são uma área de exploração ativa.

O rack de 19 polegadas (48,26 cm) ainda é a estrutura-padrão para manter servidores, apesar de esse padrão estar retornando para o hardware "railroad" dos anos 1930. Os servidores são medidos no número de unidades de rack (U) que ocupam em um rack. Um U tem 1,75 polegada (4,45 cm), e esse é o espaço mínimo que um servidor pode ocupar.

Um rack de 7 polegadas (213,36 cm) oferece 48 U, então não é coincidência que o switch mais popular para rack seja um switch ethernet de 48 portas. Esse produto se tornou uma mercadoria comum que custa cerca de US$ 30 por porta para um link ethernet de 1Gbit/s em 2011 (Barroso e Hölzle, 2009). Observe que a largura de banda dentro do rack é a mesma para cada servidor, então não importa onde o software coloca o emissor e o receptor, desde que eles estejam dentro do mesmo rack. Essa flexibilidade é ideal de uma perspectiva de software.

Esses switches geralmente oferecem 2-8 uplinks, que permitem que o rack vá para o próximo switch mais elevado na hierarquia da rede. Assim, a largura de banda deixando o rack é 6-24 vezes menor — de 48/8 para 48/2 — do que a largura de banda dentro do rack. Essa razão é chamada *oversubscription*. Infelizmente, grande oversubscription significa que os programadores devem estar cientes das consequências sobre o desempenho quando colocam emissores e receptores em racks diferentes. Esse aumento de trabalho no escalonamento de software é outro argumento para switches de rede projetados especificamente para o datacenter.

Armazenamento

Um projeto natural é preencher um rack com servidores, menos qualquer espaço de que você precisar para o switch comercial do rack ethernet. Esse projeto deixa em aberto a questão de onde o armazenamento é colocado. De uma perspectiva de construção de hardware, a solução mais simples seria incluir discos dentro do servidor, e depender da conectividade ethernet para acesso à informação nos discos de servidores remotos. A alternativa seria usar armazenamento ligado à rede (Network Attached Storage — NAS), talvez sobre uma rede de armazenamento como a Infiniband. Geralmente, a solução NAS é mais cara por terabyte de armazenamento, mas fornece muitos recursos, incluindo técnicas RAID para melhorar a confiabilidade do armazenamento.

Como você poderia esperar da filosofia expressa na seção anterior, geralmente os WSCs dependem de discos locais e fornecem software de armazenamento que lida com a conectividade e a confiabilidade. Por exemplo, GFS usa discos locais e mantém pelo menos três réplicas para superar os problemas de confiabilidade. Essa redundância cobre não só falhas de disco locais, mas também falhas de alimentação para racks e clusters inteiros. A flexibilidade de consistência eventual do GFS reduz o custo de manter as réplicas consistentes e também os requisitos de largura de banda do sistema de armazenamento. Padrões de acesso local também significam alta largura de banda para o armazenamento local, como veremos em breve.

Fique ciente de que há confusão sobre o termo *cluster* quando se fala sobre a arquitetura de um WSC. Usando a definição na Seção 6.1, um WSC é só um cluster extremamente grande. Em contraste, Barroso e Hölzle (2009) usaram o termo *cluster* para representar o próximo tamanho de agrupamento de computadores, nesse caso cerca de 30 racks. Neste capítulo, para evitar confusão, vamos usar o termo *array* para representar uma coleção de racks, preservando o significado original da palavra *cluster* para representar qualquer

elemento de uma coleção de computadores em rede dentro de um rack a um depósito de computadores em rede.

Switch de arrays

O switch que conecta um array de racks é consideravelmente mais caro do que o switch ethernet comercial de 48 portas. Esse custo é devido, em parte, à maior conectividade e em parte ao fato de a largura de banda através do switch ser muito maior para reduzir o problema de oversubscription. Barroso e Hölzle (2009) relataram que um switch com 10 vezes a *largura de banda de bisseção* — basicamente, a largura de banda interna no pior caso — de um switch de rack custa cerca de 100 vezes mais. Uma das razões disso é que o custo da largura de banda de switch para n portas pode aumentar na proporção n^2.

Outra razão para os altos custos é que esses produtos oferecem altas margens de lucro para as empresas que os produzem. Eles justificam tais preços em parte fornecendo características como inspeção de pacote, que são caras porque operam a taxas muito altas. Por exemplo, switches de rede são grandes usuários de chips de memória endereçáveis por conteúdo e de gate-arrays programáveis (Field Programmable Gate Arrays — FPGAs) que fornecem esses recursos, mas os próprios chips são caros. Embora tais recursos possam ser valiosos para configurações de internet, geralmente eles são mal utilizados dentro do datacenter.

Hierarquia de memória de WSC

A Figura 6.6 mostra a latência, a largura de banda e a capacidade da hierarquia de memória dentro de um WSC, e a Figura 6.7 mostra os mesmos dados visualmente. Essas figuras são baseadas nas seguintes suposições (Barroso e Hölzle, 2009):

- Cada servidor contém 16 GBytes de memória com tempo de acesso de 100 nanossegundos e a transferência é a 20 GBytes/s, e 2 terabytes de disco com tempo de acesso de 10 milissegundos e a transferência é a 200 MBytes/s. Existem dois soquetes por placa, e eles compartilham uma porta ethernet de 1 Gbit/s.
- Cada par de racks inclui um switch de rack e contém 80 servidores 2U (Seção 6.7). O software de rede mais o overhead de switch aumentam a latência da DRAM para 100 microssegundos e a latência de acesso de disco para 11 milissegundos. Assim, a capacidade total de armazenamento de um rack é de aproximadamente 1 terabyte de DRAM e 160 terabytes de armazenamento de disco. A ethernet de 1 Gbit/s limita a largura de banda remota para a DRAM ou disco dentro do rack para 100 MBytes/s.
- O switch de array pode tratar 30 racks, então a capacidade de armazenamento de um array aumenta por um fator de 30:30 terabytes de DRAM e 4,8 petabytes

	Local	Rack	Array
Latência de DRAM (microssegundos)	0,1	100	300
Latência de disco (microssegundos)	10.000	11.000	12.000
Largura de banda de DRAM (MB/s)	20.000	100	10
Largura de banda de disco (MB/s)	200	100	10
Capacidade de DRAM (GB)	16	1.040	31.200
Capacidade de disco (GB)	2.000	160.000	4.800.000

FIGURA 6.6 Latência, largura de banda e capacidade da hierarquia de memória de um WSC (Barroso e Hölzle, 2009).
A Figura 6.7 mostra a mesma informação.

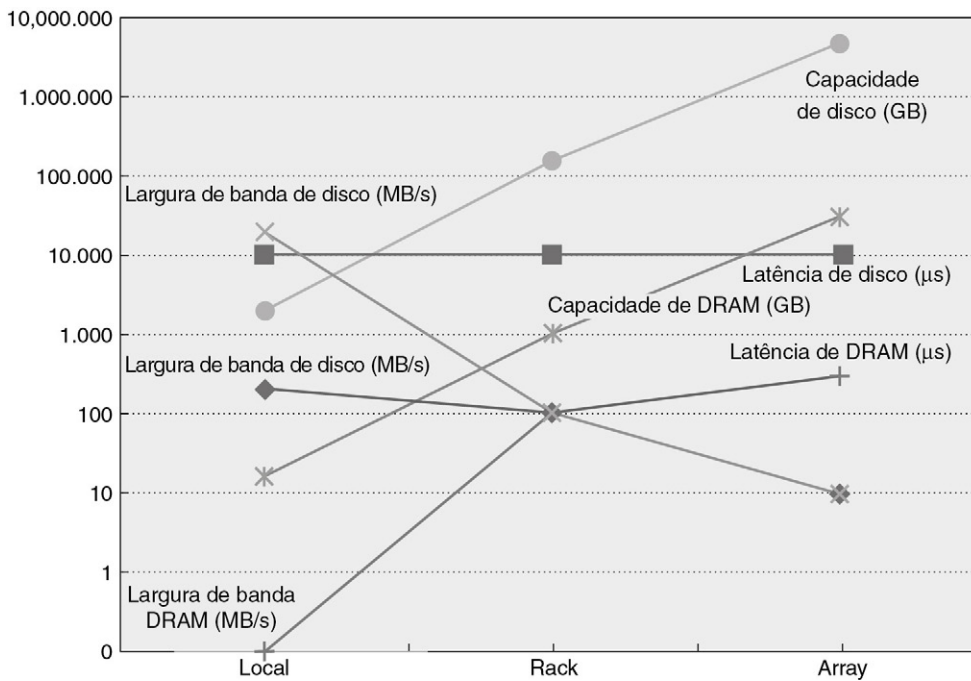



FIGURA 6.7 Gráfico de latência, largura de banda e capacidade da hierarquia de memória de um WSC para os dados da Figura 6.6 (Barroso e Hölzle, 2009).

de disco. O hardware do switch de array e o software aumentam a latência da DRAM dentro de um array para 500 microssegundos e a latência de disco para 12 milissegundos. A largura de banda do switch de array limita a largura de banda remota para a DRAM de array e disco de array para 10 MBytes/s.

As Figuras 6.6 e 6.7 mostram que o overhead de rede aumenta drasticamente a latência da DRAM local para o rack DRAM e array DRAM, mas os dois têm latência mais de 10 vezes melhor do que o disco local. A rede diminui a diferença entre a DRAM de rack e o disco de rack e entre a DRAM de array e o disco de array.

O WSC precisa de 20 arrays para atingir 50.000 servidores, então há mais um nível na hierarquia de rede. A Figura 6.8 mostra os rotadores convencionais de nível 3 para conectar os arrays e para a internet.

A maioria das aplicações cabe em um único array dentro de um WSC. Aqueles que precisam de mais de um array usam *sharding* ou *particionamento*, o que significa que o conjunto de dados é dividido em partes independentes e então distribuídos entre arrays diferentes. Operações sobre todo o conjunto de dados são enviadas para os servidores que hospedam as partes, e os resultados são reunidos pelo computador cliente.

Exemplo Qual é a latência média de memória supondo que 90% de acessos são locais ao servidor, 9% estão fora do servidor, mas dentro do rack, e 1% está fora do rack, mas dentro do array?

Resposta O tempo médio de acesso à memória é

$$(90\% \times 0{,}1) + (9\% \times 100) + (1\% \times 300) = 0{,}09 + 9 + 3 = 12{,}09 \text{ microssegundos}$$

Ou um fator de mais de 120 de redução de velocidade *versus* 100% de acessos locais. Obviamente, a localidade de acesso dentro de um servidor é vital para o desempenho de um WSC.

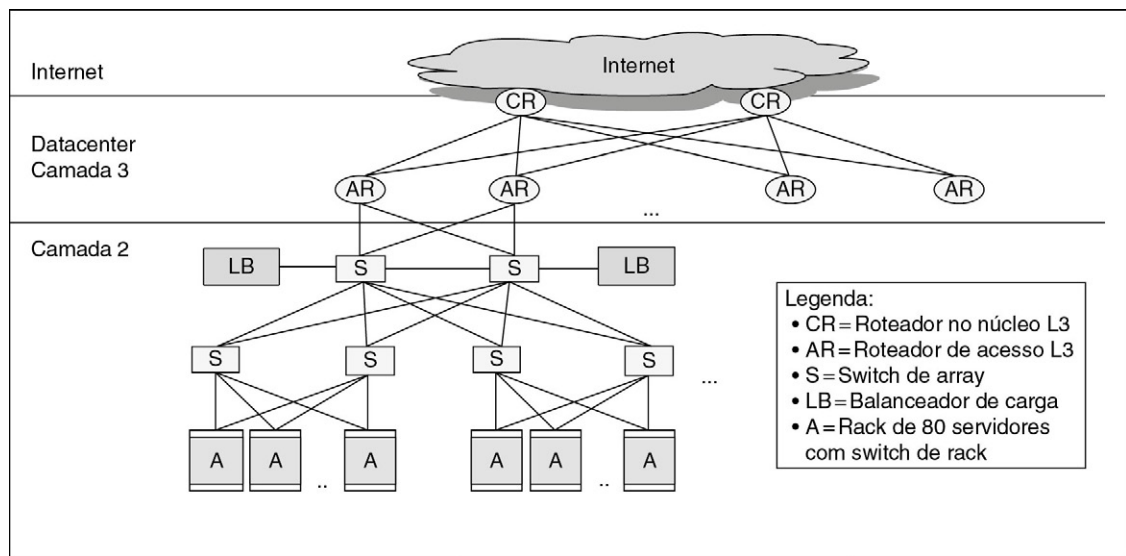



FIGURA 6.8 **A rede de camada 3 usada para ligar arrays entre si e para a internet (Greenberg *et al.*, 2009).**
Alguns WSCs usam um *roteador de limite* separado para conectar a internet aos switches de nível 3 do datacenter.

Exemplo Quanto tempo leva para transferir 1.000 MB entre discos dentro do servidor, entre servidores no rack e entre servidores em racks diferentes no array? Quão mais rápido é transferir 1.000 MB entre DRAM nesses três casos?

Resposta Uma transferência de 1.000 MB entre discos leva:

$$\text{Dentro do servidor} = 1.000 / 200 = 5 \text{ segundos}$$

$$\text{Dentro do rack} = 1.000 / 100 = 10 \text{ segundos}$$

$$\text{Dentro do array} = 1.000 / 10 = 100 \text{ segundos}$$

Uma transferência de bloco memória a memória leva:

$$\text{Dentro do servidor} = 1.000 / 20.000 = 0{,}05 \text{ segundo}$$

$$\text{Dentro do rack} = 1.000 / 100 = 10 \text{ segundos}$$

$$\text{Dentro do array} = 1.000 / 10 = 100 \text{ segundos}$$

Assim, para transferências de bloco fora de um único servidor, não importa nem mesmo se os dados estão na memória ou no disco, já que o switch de rack e o switch de array são os gargalos. Esses limites de desempenho afetam o projeto de software WSC e inspiram a necessidade de switches de maior desempenho (Seção 6.6).

Dada a arquitetura do equipamento de TI, estamos agora prontos para ver como armazenar, alimentar e refrigerá-lo e para discutir o custo de construir e operar todo o WSC, somente em comparação ao equipamento de TI dentro dele.

6.4 INFRAESTRUTURA FÍSICA E CUSTOS DOS COMPUTADORES EM ESCALA WAREHOUSE

Para construir um WSC, primeiro você precisa construir um depósito. Uma das primeiras questões é: "Onde?" Os agentes imobiliários enfatizam a localização, mas a localização para um WSC quer dizer proximidade a um backbone de internet, fibras ópticas, baixo custo da eletricidade e baixo risco de desastres ambientais, como terremotos, enchentes e furacões. Para uma empresa com muitos WSCs, outra preocupação é encontrar um local

geograficamente próximo de uma população atual ou futura de usuários de internet, de modo a reduzir a latência na internet. Há também preocupações muito mais mundanas, como impostos sobre propriedade.

Os custos de infraestrutura para distribuição de energia e refrigeração são muito maiores do que os custos de concentração de um WSC, então nos concentramos nos primeiros. As Figuras 6.9 e 6.10 mostram a distribuição de energia e infraestrutura de refrigeração dentro de um WSC.

Embora haja muitas variações implementadas, na América do Norte a energia elétrica geralmente passa por cinco etapas e quatro mudanças de voltagem no caminho para o servidor, começando com as linhas de alta voltagem na torre de transmissão de 115.000 volts:

1. A subestação muda de 115.000 volts para linhas de média voltagem de 13.200 volts, com uma eficiência de 99,7%.
2. Para impedir que todo o WSC saia do ar se a energia acabar, um WSC conta com um fornecimento de energia ininterrupto (UPS), assim como alguns servidores. Nesse caso, ele envolve grandes motores a diesel que podem substituir a companhia elétrica em uma emergência, e baterias ou flywheels para manter a energia depois que o fornecimento for perdido, mas antes que os motores a diesel estejam prontos. Os geradores e baterias podem ocupar tanto espaço que geralmente são localizados em uma sala separada do equipamento de TI. A UPS tem três funções: condicionamento de potência (manter níveis apropriados de tensão e outras características), mantendo a carga elétrica enquanto os geradores são inicializados e entram em linha, e mantendo a carga elétrica ao voltar dos geradores para a rede elétrica. A eficiência dessa grande UPS é de 94%, então a instalação perde 6% da alimentação por ter uma UPS. A UPS do WSC pode responder por 7-12% do custo de todo o equipamento de TI.

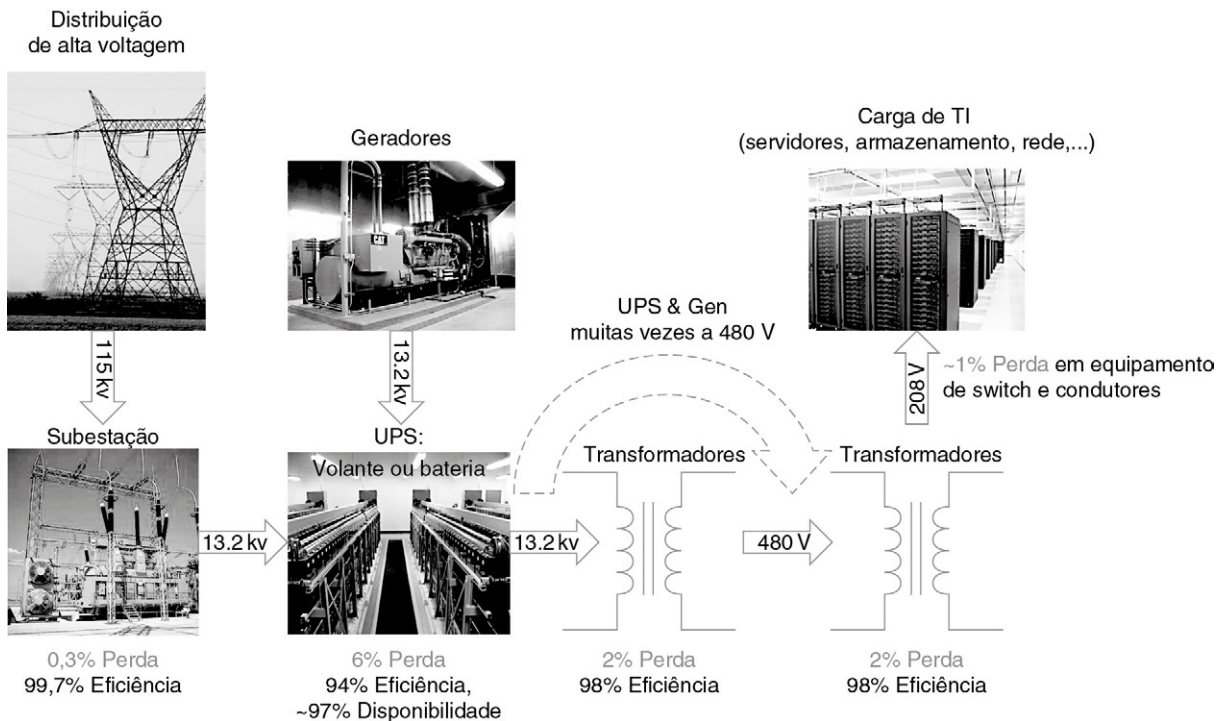



FIGURA 6.9 Distribuição de energia e onde as perdas ocorrem.
Observe que o melhor desempenho é de 11%. *(De Hamilton, 2010.)*

3. A seguir no sistema está uma unidade de distribuição de energia (Power Distribution Unit — PDU) que converte para energia de baixa voltagem, interna e trifásica, a 480 volts. A eficiência da conversão é de 98%. Uma PDU típica lida com 75-225 kilowatts de carga, ou cerca de 10 racks.
4. Há ainda mais uma etapa de redução para energia bifásica a 208 volts que os servidores podem usar, novamente com 98% de eficiência. (Dentro do servidor, há mais etapas para reduzir a voltagem para uma que os chips possam usar; Seção 6.7.)
5. Conectores, breakers e fiação elétrica para o servidor têm eficiência coletiva de 99%.

OS WSC fora da América do Norte usam valores de conversão diferentes, mas o projeto geral é o mesmo.

Juntando tudo, a eficiência de transformar uma alimentação de 115.000 volts da rede em 208 volts que os servidores possam usar é de 89%.

$$99,7\% \times 94\% \times 98\% \times 98\% \times 99\% = 89\%$$

Essa eficiência geral deixa um pouco mais de 10% de folga para melhoria, mas, como veremos, os engenheiros ainda estão tentando melhorar isso.

Há consideravelmente mais oportunidade de melhoria na infraestrutura de refrigeração. A unidade de ar condicionado da sala do computador (Computer Room Air Conditioning — CRAC) resfria o ar na sala do servidor usando água refrigerada, de modo similar ao de um refrigerador removendo calor e liberando-o fora do refrigerador. Quando um líquido absorve calor, ele evapora. Ao contrário, quando um líquido libera calor, ele condensa. Aparelhos de ar-condicionado bombeiam o líquido em espirais sob baixa pressão para evaporar e absorver calor, que é então enviado para um condensador externo onde é liberado. Assim, em uma unidade CRAC, ventoinhas empurram o ar para além de um conjunto de espirais preenchidas com água fria e uma bomba move a água aquecida para os resfriadores externos para ser refrigerada. Geralmente, o ar frio para servidores fica entre 18-22 °C (entre 64-71 °F). A Figura 6.10 mostra a grande coleção de ventoinhas e bombas de água que movem ar e água através do sistema.

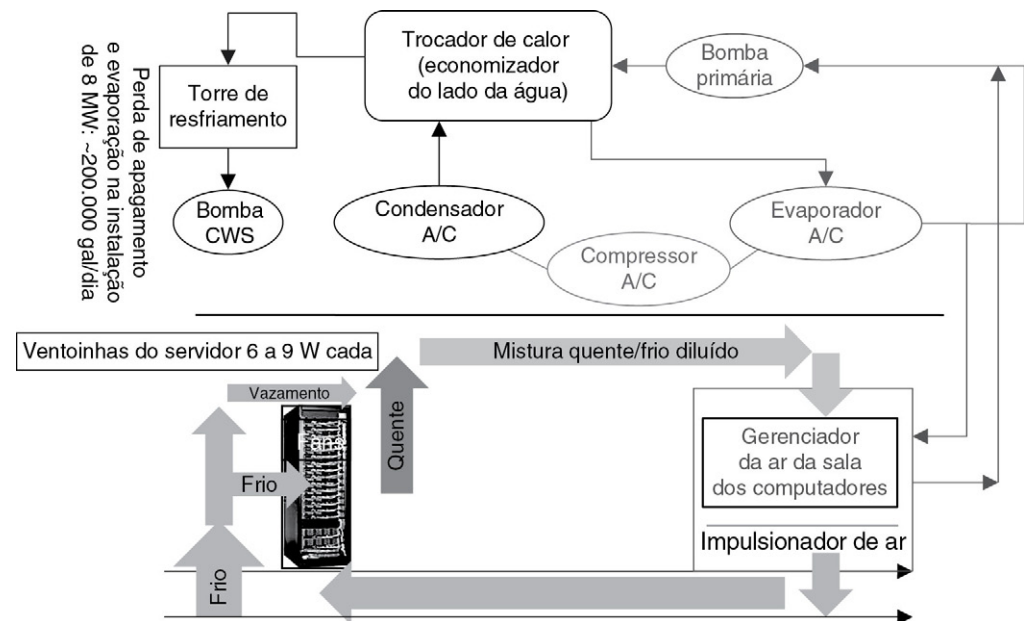



FIGURA 6.10 Projeto mecânico para sistema de refrigeração.
CWS significa Sistema de Circulação de Água (Circulating Water System). *(De Hamilton, 2010.)*

Obviamente, um dos modos mais simples de melhorar a eficiência energética é fazer com que o equipamento de TI funcione a temperaturas mais altas de modo que o ar não precise ser tão resfriado. Alguns WSCs rodam seu equipamento consideravelmente acima de 22 °C (71 °F).

Além dos resfriadores, torres de resfriamento são usadas em alguns datacenters para aproveitar o ar externo mais frio para refrigerar a água antes de enviá-la para os resfriadores. A temperatura que importa é chamada *temperatura wet-bulb*. Ela é medida soprando-se ar na extremidade em bulbo de um termômetro que contenha água. Essa é a temperatura mais baixa que pode ser atingida ao evaporar água com ar.

A água quente flui ao longo de uma grande superfície na torre, transferindo calor para o ar externo através de evaporação; isso resfria a água. Essa técnica é chamada *economia airside*. Uma alternativa é usar água fria em vez de ar frio. O WSC do Google na Bélgica usa um intercooler água a água que usa água fria de um canal industrial para resfriar a água quente de dentro do WSC.

O fluxo de ar é cuidadosamente planejado para o próprio equipamento de TI — alguns projetos usam até simuladores de fluxo de ar. Projetos eficientes preservam a temperatura no ar frio, reduzindo as chances de ele se misturar com ar quente. Por exemplo, um WSC pode ter ilhas alternadas de ar quente e frio orientando os servidores em direções opostas, de modo que a exaustão ocorre em direções alternadas.

Além das perdas de energia, o sistema de refrigeração também usa muita água devido à evaporação ou a vazamentos ao longo das tubulações. Por exemplo, uma instalação de 8 MW pode usar de 70.000-200.000 galões de água por dia.

Os custos relativos de energia do equipamento de refrigeração para equipamentos de TI em um datacenter típico (Barroso e Hölzle, 2009) são os seguintes:

- Os resfriadores respondem por 30-50% do consumo de energia do equipamento de TI.
- O CRAC responde por 10-20% do consumo de energia do equipamento de TI, devido principalmente às ventoinhas.

Surpreendentemente, não é óbvio entender quantos servidores um WSC pode suportar depois que se subtraem os overheads para distribuição e refrigeração de água. O chamado *nameplate power rating* do fabricante do servidor é sempre conservador. É a energia máxima que um servidor pode consumir. A primeira etapa, portanto, é medir um único servidor sob uma variedade de cargas de trabalho a serem empregadas no WSC. (Em geral, a rede é responsável por cerca de 5% do consumo de energia, então pode ser ignorada no começo.)

Para determinar o número de servidores para um WSC, a energia disponível para TI poderia ser simplesmente dividida pelo consumo de energia medido por servidor. Entretanto, isso seria novamente muito conservador de acordo com Fan, Weber e Barroso (2007). Eles descobriram que há uma lacuna significativa entre o que milhares de servidores poderiam fazer teoricamente no pior caso e o que eles fazem na prática, já que nenhuma carga de trabalho real vai manter milhares de servidores simultaneamente em seus picos. Eles descobriram que poderiam ultrapassar com segurança o número de servidores em até 40% com base no consumo de energia de um único servidor. Recomendaram que os arquitetos de WSC deveriam fazer isso para aumentar a utilização média de energia dentro de um WSC. Entretanto, eles também sugeriram usar monitoramento extensivo de software em conjunto com um mecanismo de segurança que desescalona as tarefas de baixa prioridade no caso de a carga de trabalho mudar.

Discriminando o uso de energia dentro do próprio equipamento de TI, Barroso e Hölzle (2009) reportaram o seguinte para um WSC do Google implementado em 2007:

- 33% de energia para os processadores
- 30% para a DRAM
- 10% para os discos
- 5% para a rede
- 22% para outros recursos (dentro do servidor)

Medindo a eficiência de um WSC

Uma métrica simples e amplamente usada para avaliar a eficiência de um datacenter ou um WSC é chamada *efetividade de utilização de energia* (Power Utilization Effectiveness — PUE):

PUE = (consumo total de energia da instalação) / (consumo de energia do equipamento de TI)

Assim, o PU deve ser maior ou igual a 1 e, quanto maior o PUE, menos eficiente o WSC.

Greenberg *et al*. (2006) relataram o PUE de 19 datacenters e a parte do overhead que foi para a infraestrutura de refrigeração. A Figura 6.11 mostra o que eles descobriram, ordenado do PUE mais eficiente para o menos eficiente. O PUE mediano é de 1,69, com a infraestrutura de refrigeração usando mais de metade da energia dos próprios servidores — em média, 0,55 do 1,69 destina-se a refrigeração. Observe que essas PUEs são médias e podem variar diariamente, dependendo da carga de trabalho e até mesmo da temperatura do ar externo, como nós veremos.

Como o desempenho por dólar é a métrica final, ainda precisamos medir o desempenho. Como mostra a Figura 6.7, a largura de banda cai e a latência aumenta, dependendo da distância até os dados. Em um WSC, a largura de banda de DRAM dentro de um servidor é 200 vezes maior do que dentro de um rack, que, por sua vez, é 10 vezes maior do que dentro de um array. Assim, há outro tipo de localidade para considerar no posicionamento de dados e programas dentro de um WSC.

Enquanto os projetistas de um WSC muitas vezes se concentram na largura de banda, os programadores que desenvolvem aplicações em um WSC também se preocupam com a latência, já que ela é visível para os usuários. A satisfação e a produtividade dos usuários estão ligadas ao tempo de resposta de um serviço. Muitos estudos dos dias de compartilhamento de tempo informam que a produtividade do usuário é inversamente proporcional ao tempo de uma interação, em geral discriminada em tempo de entrada humana, tempo de resposta do sistema e tempo para a pessoa pensar sobre a resposta antes de inserir a próxima entrada. Os resultados dos experimentos mostraram que reduzir o tempo de resposta de sistema em 30% reduzia o tempo de uma interação em 70%. Esse resultado implausível foi explicado pela natureza humana: as pessoas precisam de menos tempo para pensar quando recebem uma resposta rápida, já que é menos provável que se distraiam.

A Figura 6.12 mostra os resultados de tal experimento para o sistema de busca Bing, em que atrasos de 50-2.000 ms foram inseridos no servidor de busca. Como esperado a partir dos estudos anteriores, o tempo até o próximo clique praticamente dobrou o atraso. Ou seja, o atraso de 200 ms no servidor levou a um aumento de 500 ms no tempo até o próximo clique. O rendimento caiu linearmente com o aumento do atraso, assim como a satisfação do cliente. Um estudo realizado no serviço de buscas Google descobriu que esses efeitos duraram muito tempo depois que o experimento de quatro semanas acabou. Cinco semanas mais tarde, havia 0,1% menos buscas por dia para usuários que experimentaram atrasos de 200 ms, e menos 0,2% buscas de usuários que experimentaram atrasos de 400 ms. Dada a quantidade de dinheiro ganha com busca, até mesmo mudanças tão pequenas

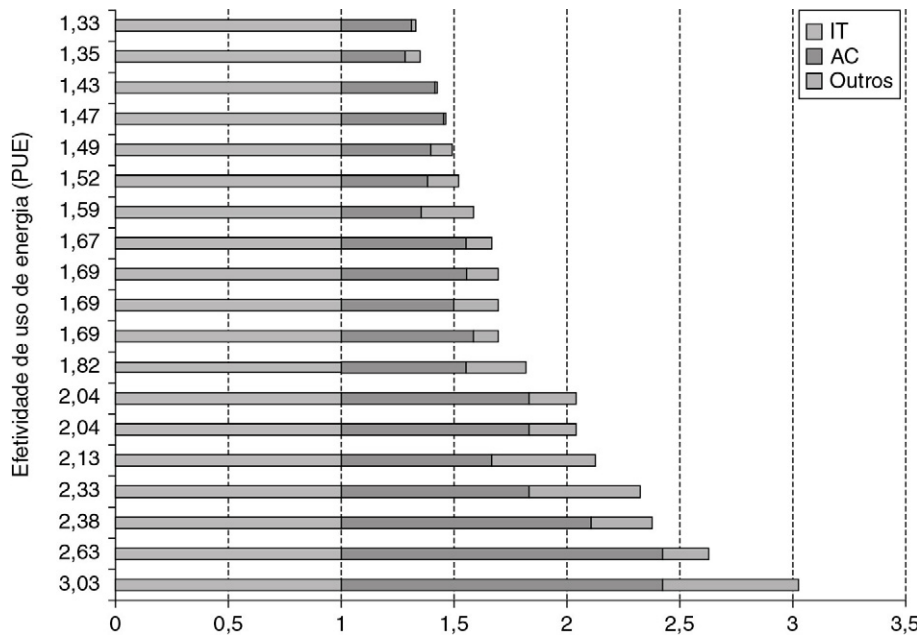



FIGURA 6.11 **Eficiência de utilização energética de 19 datacenters em 2006 (Greenberg *et al.*, 2006).**
O consumo de energia para ar-condicionado (AC) e outros usos (como distribuição de energia) é normalizado para a energia para o equipamento de TI no cálculo do PUE. Assim, a energia para equipamento de TI poderia ser 1,0, e AC varia de cerca de 0,30-1,40 vez o consumo de energia do equipamento de TI. O consumo de energia para "outros" varia de 0,05-0,60 do equipamento de TI.

são desconcertantes. De fato, os resultados foram tão negativos que eles encerraram o experimento prematuramente (Schurman e Brutlag, 2009).

Devido a essa preocupação extrema com a satisfação de todos os usuários de um serviço de internet, geralmente os objetivos de desempenho são especificados com alta porcentagem de requisição estando abaixo de um limite de latência em vez de somente oferecer um objetivo para a latência média. Tais objetivos de limite são chamados *objetivos de nível de serviço* (Service Level Objectives — SLOs) ou *acordos de nível de serviço* (Service Level Agreements — SLAs). Em SLO pode ser que 99% das requisições ocorram em menos de 100 milissegundos. Assim, os projetistas do sistema de armazenamento Dynamo da Amazon decidiram que, para que os serviços oferecessem boa latência sobre o Dynamo, seu sistema de armazenamento deveria funcionar com o objetivo de latência de 99,9% do tempo (DeCandia *et al.*, 2007). Por exemplo, uma melhoria no Dynamo ajudou o 99,9° ponto porcentual muito mais no caso médio, que reflete suas prioridades.

Atraso de servidor (ms)	Aumento no tempo para o próximo clique (MS)	Buscas/ usuário	Qualquer clique/usuário	Satisfação do usuário	Receita/ usuário
50	–	–	–	–	–
200	500	–	–0,3%	–0,4%	–
500	1.200	–	–1,0%	–0,9%	–1,2%
1.000	1.900	–0,7%	–1,9%	–1,6%	–2,8%
2.000	3.100	–1,8%	–4,4%	–3,8%	–4,3%

FIGURA 6.12 **Impacto negativo dos atrasos no sistema de busca Bing sobre o comportamento do usuário (Schurman e Brutlag, 2009).**

Custo de um WSC

Como mencionado na introdução, ao contrário da maioria dos arquitetos, os projetistas de WSCs se preocupam com os custos operacionais, além do custo para construir o WSC. A contabilidade chama os primeiros custos de *despesas operacionais* (OPEX) e os últimos de *despesas capitais* (CAPEX).

Para colocar os custos com energia em perspectiva, Hamilton (2010) realizou um estudo de caso para estimar os custos de um WSC. Ele determinou que o CAPEX dessa instalação de 8 MW era de US$ 88 milhões e que os cerca de 46.000 servidores e equipamento de rede correspondente somavam outros US$ 79 milhões para o CAPEX do WSC. A Figura 6.13 mostra o restante das suposições para o estudo de caso.

Podemos agora estabelecer o custo total da energia, já que as regras de contabilidade dos Estados Unidos nos permitem converter CAPEX em OPEX. Podemos simplesmente amortizar o CAPEX como uma quantia fixa a cada mês para a vida efetiva do equipamento. A Figura 6.14 discrimina o OPEX mensal para esse estudo de caso. Observe que as taxas de amortização diferem significativamente de 10 anos para a instalação a quatro anos para o equipamento de rede e três anos para os servidores. Portanto, a instalação WSC dura

Tamanho da instalação (carga crítica em watts)	8.000.000
Uso médio de energia (%)	80%
Efetividade do uso de energia	1.45
Custo da energia ($ /kWh)	$ 0,07
% da infraestrutura energética e de refrigeração (% do custo total da instalação)	82%
CAPEX para a instalação (não incluindo equipamento de TI)	**$ 88.000.000**
Número de servidores	45.978
Custo/servidor	$ 1.450
CAPEX para os servidores	**$ 66.700.000**
Número de switches de rack	1150
Custo/switch de rack	$ 4.800
Número de switches de array	22
Custo/switch de array	$ 300.000
Número de switches de nível 3	2
Custo/switch de nível 3	$ 500.000
Número de roteadores de limite	2
Custo/roteador de limite	$ 144.800
CAPEX para equipamento de rede	**$ 12.810.000**
CAPEX total para WSC	**$ 167.510.000**
Tempo de amortização do servidor	3 anos
Tempo de amortização de rede	4 anos
Tempo de amortização da instalação	10 anos
Custo anual do dinheiro	5%

FIGURA 6.13 Estudo de caso para um QSC, baseado em Hamilton (2010) arredondado para os US$ 5.000 mais próximos.

Os custos de largura de banda de internet variam por aplicação, por isso eles não são incluídos aqui. Os 18% restantes do CAPEX para a instalação incluem comprar a propriedade e o custo da construção do edifício. Nós adicionamos os custos de pessoal para administração de segurança e instalações na Figura 6.14, que não eram parte do estudo de caso. Observe que as estimativas de Hamilton foram feitas antes de ele se unir à Amazon, e não são baseadas no WSC de uma empresa em particular.

Gasto (% do total)	Categoria	Custo mensal	Porcentagem do custo mensal
CAPEX amortizado (85%)	Servidores	$ 2.000.000	53%
	Equipamento de rede	$ 290.000	8%
	Infraestrutura energética e de refrigeração	$ 765.000	20%
	Outras infraestruturas	$ 170.000	4%
OPEX (15%)	Uso mensal de energia	$ 475.000	13%
	Salários e benefícios mensais do pessoal	$ 85.000	2%
	OPEX total	$ 3.800.000	100%

FIGURA 6.14 **OPEX mensal para a Figura 6.13, arredondado para os US$ 5.000 mais próximos.**
Observe que a amortização de três anos para servidores significa que você precisa comprar novos servidores a cada três anos, enquanto a instalação é amortizada em 10 anos. Portanto, os custos amortizados de capital para os servidores são cerca de três vezes maiores do que para a instalação. Os custos com pessoal incluem três guardas de segurança continuamente 24 horas por dia, 365 dias por ano, a US$ 20 por hora por pessoa, e uma pessoa de instalações 24 horas por dia, 365 dias por ano, a US$ 30 por hora. Os benefícios são 30% dos salários. Esse cálculo não inclui o custo de largura de banda para a internet, já que ele varia por aplicação, nem taxas de manutenção do fornecedor, já que elas variam por equipamento e negociação.

uma década, mas você precisa substituir os servidores a cada três anos e o equipamento de rede a cada quatro anos. Ao amortizar o CAPEX, Hamilton descobriu um OPEX mensal incluindo o custo de empréstimos monetários (5% anualmente) para pagar pelo WSC. A US$ 3,8 milhões por mês, o OPEX é de cerca de 2% do CAPEX.

Essa figura nos permite calcular um guia útil para ter em mente ao tomarmos decisões sobre quais componentes usar quando estamos preocupados com a energia. O custo totalmente considerado de um watt por ano em um WSC, incluindo o custo de amortizar a infraestrutura energética e de refrigeração, é

$$\frac{\text{Custo mensal da infraestrutura} + \text{custo mensal da energia}}{\text{Tamanho da instalação em watts}} \times 12 = \frac{\$765\text{k} + \$475\text{k}}{8\text{M}} \times 12 = \$1{,}86$$

O custo é de cerca de US$ 2 por watt-ano. Assim, para reduzir custos através da economia de energia, você não deve gastar mais de US$ 2 por watt-ano (Seção 6.8).

Observe que mais de um terço do OPEX se relaciona à energia, com essa categoria tendendo a subir enquanto os custos de servidor tendem a descer ao longo do tempo. O equipamento de rede é significativo com 8% do OPEX total e 19% do CAPEX do servidor, e o equipamento de rede não tende a descer tão rapidamente quanto os servidores. Essa diferença é especialmente verdadeira para os switches na hierarquia de rede acima do rack, que representam mais dos custos de rede (Seção 6.6). Os custos com pessoal para administração da segurança e instalações correspondem a somente 2% do OPEX. Dividindo o OPEX na Figura 6.14 pelo número de servidores e horas por mês, o custo é de cerca de US$ 0,11 por servidor por hora.

Exemplo O custo da eletricidade varia, por região nos Estados Unidos, de US$ 0,03-0,15 por kilowatt-hora. Qual é o impacto dos custos horários de servidor dessas duas taxas extremas?

Resposta Nós multiplicamos a carga crítica de 8 MW pelo PUE e pelo uso médio de energia da Figura 6.13 para calcular o uso médio de energia:

$$8 \times 1{,}45 \times 80\% = 9{,}28 \text{ megawatts}$$

Portanto, o custo mensal pela energia vai de US$ 475.000, na Figura 6.14, para US$ 205.000 a US$ 0,03 por kilowatt-hora e para US$ 1.015.000 a US$ 0,15 por kilowatt-hora. Essas mudanças no custo da eletricidade mudam os custos horários de servidor de US$ 0,11 para US$ 0,10 e US$ 0,13, respectivamente.

Exemplo O que aconteceria com os custos mensais se os tempos de amortizações fossem todos os mesmos — cinco anos, por exemplo? Como isso mudaria o custo horário por servidor?

Resposta A planilha está disponível on-line em http://mvdirona.com/jrh/TalksAndPapers/ PerspectivesDataCenterCostAndPower.xls. Mudando o tempo de amortização para cinco anos, mudam as quatro primeiras linhas da Figura 6.14 para

Servidores	$ 1.260.000	37%
Equipamento de rede	$ 242.000	7%
Infraestrutura energética e de refrigeração	$ 1.115.000	33%
Outras infraestruturas	$ 245.000	7%

E o OPEX mensal total é de US$ 3.422.000. Se substituíssemos tudo a cada cinco anos, o custo seria de US$ 0,103 por hora de servidor, com a maior parte dos custos amortizados para a instalação e não para os servidores, como na Figura 6.14.

A taxa de US$ 0,11 por servidor por hora pode ser muito menor do que o custo para muitas empresas que são proprietárias e operam seus próprios (menores) datacenters convencionais. A vantagem de custo dos WSCs levou grandes empresas de internet a oferecerem a computação como serviço pelo qual, como a eletricidade, você paga somente pelo que usa. Hoje, a computação como serviço é mais conhecida como computação em nuvem.

6.5 COMPUTAÇÃO EM NUVEM: O RETORNO DA COMPUTAÇÃO DE UTILIDADE

> Se os computadores do tipo que eu defendi se tornarem os computadores do futuro, então algum dia a computação poderá ser organizada como um serviço público, assim como o sistema telefônico […] O serviço de computadores pode se tornar a base de uma nova e importante indústria.
>
> **John McCarthy, *Celebração do centenário do MIT* (1961)**

Impulsionadas pela demanda de um número cada vez maior de usuários, as empresas de internet, como Amazon, Google e Microsoft, construíram computadores em escala warehouse maiores a partir de componentes comerciais comuns. Essa demanda levou a inovações em softwares de sistema para suportar a operação nessa escala, incluindo Bigtabel, Dynamo, GFS e MapReduce. Ela também exigiu melhorias nas técnicas operacionais para entregar um serviço disponível pelo menos 99,99% do tempo, apesar das falhas de componentes e dos ataques à segurança. Exemplos dessas técnicas incluem failover, firewalls, máquinas virtuais e proteção contra ataques distribuídos de denial-of-service. Com o software e a experiência proporcionando a capacidade de escalar e aumentar a demanda dos clientes que justificou o investimento, os WSCs com 50.000-100.000 servidores se tornaram comuns em 2011.

Com a maior escala vieram maiores economias de escala. Com base em um estudo de 2006, que comparou um WSC com um datacenter com somente 1.000 servidores, Hamilton (2010) reportou as seguintes vantagens:

- *Redução de 5,7 vezes nos custos de armazenamento.* Ele custa ao WSC US$ 4,6 por GByte por ano para armazenamento em disco *versus* US$ 26 por GByte para o datacenter.

- *Redução de 7,1 vezes nos custos administrativos.* A razão de servidores por administrador foi de mais de 1.000 para o WSC *versus* somente 140 para o datacenter.
- *Redução de 7,3 vezes nos custos de rede.* A largura de banda de internet custa ao WSC US$ 13 por Mbit/s/mês contra US$ 95 para o datacenter. Não é de surpreender que você possa negociar um preço muito melhor por Mbit/s se contratar 1.000 Mbit/s do que se contratar 10 Mbit/s.

Outra economia de escala vem durante a compra. O alto nível de compras leva a preços com desconto por volume nos servidores e no equipamento de rede. Ele também leva à otimização da cadeia de fornecimento. Dell, IBM e SGI ainda entregam pedidos novos a um WSC em uma semana em vez de 4-6 meses. Um tempo de entrega curto torna mais fácil aumentar o serviço para atender à demanda.

As economias de escala também se aplicam aos custos operacionais. A partir da seção anterior, vimos que muitos datacenters operam com um PUE de 2,0. Grandes empresas podem justificar contratar engenheiros mecânicos e de energia para desenvolver WSCs com PUEs mais baixos, na faixa de 1,2 (Seção 6.7).

Os serviços de internet precisam ser distribuídos para múltiplos WSCs para garantir confiabilidade e para reduzir a latência, especialmente nos mercados internacionais. Todas as grandes empresas usam múltiplos WSCs por esse motivo. Para empresas individuais, é muito mais caro criar diversos pequenos datacenter ao redor do mundo do que um único datacenter na matriz.

Finalmente, pelas razões apresentadas na Seção 6.1, os servidores nos datacenter tendem a ser utilizados somente 10-20% do tempo. Ao tornar os WSCs disponíveis ao público, picos não correlacionados entre diferentes clientes podem gerar uma utilização média acima de 50%.

Assim, as economias de escala para um WSC oferecem fatores de 5-7 para diversos componentes de um WSC, além de alguns fatores de 1,5-2 para todo o WSC.

Embora existam muitos provedores de computação em nuvem, nós apresentamos o Amazon Web Services (AWS) em parte devido à sua popularidade e noutra devido à abstração de baixo nível e, portanto, mais flexível de seu serviço. O Google App Engine e o Microsoft Azure elevam o nível de abstração para o gerenciamento do tempo de execução e para oferecer serviços automáticos de escalamento, que atendem melhor a alguns clientes, mas não bom quanto o AWS para o conteúdo deste livro.

Amazon Web Services

A computação como serviço vem desde os sistemas comerciais de compartilhamento de tempo e mesmo os sistemas de processamento de lote dos anos 1960 e 1970, em que as empresas só pagavam por um terminal e uma linha telefônica e eram cobradas com base em quanta computação usavam. Muitos esforços desde o fim do compartilhamento de tempo vêm tentando oferecer tais serviços "pagamento conforme o uso", mas muitas vezes eles não têm sucesso.

Quando a Amazon começou a oferecer computação como serviço através do Amazon Simple Storage Service (Amazon S3) e do Amazon Elastic Computer Cloud (Amazon EC2) em 2006, tomou algumas decisões técnicas e de negócio inovadoras.

- *Máquinas virtuais:* Construir o WSC usando computadores x86 comeerciais executando o sistema operacional Linux e a máquina virtual Xen solucionou muitos problemas: 1) permitiu à Amazon proteger os usuários uns dos outros;

2) simplificou a distribuição de software dentro de um WSC, no sentido de que os clientes só precisam instalar uma imagem e então o AWS vai automaticamente distribuí-la para todas as instâncias sendo usadas; 3) a capacidade de "matar" uma máquina virtual com confiabilidade torna fácil para a Amazon e para os clientes controlarem o uso dos recursos; 4) uma vez que as máquinas virtuais podem limitar a taxa na qual eles usam os processadores físicos, os discos e a rede além da quantidade de memória principal, o que deu ao AWS diversas opções de preço, a opção de preço mais baixo empacotando diversos núcleos virtuais em um único servidor, a opção de preço mais alta de acesso exclusivo a todos os recursos da máquina, além de diversos pontos intermediários; 5) as máquinas virtuais ocultam a identidade de hardwares mais antigos, permitindo ao AWS continuar a vender tempo em máquinas mais velhas, que poderiam não ser atrativas para os clientes se eles soubessem sua idade; 6) as máquinas virtuais permitem ao AWS apresentar hardware novo e mais rápido, empacotando ainda mais núcleos virtuais por servidor ou simplesmente oferecendo instâncias que tenham maior desempenho por núcleo virtual. *Virtualização* significa que o desempenho oferecido não precisa ser um múltiplo inteiro do desempenho do hardware.

- *Custo muito baixo.* Quando o AWS anunciou uma taxa de US$ 0,10 por hora por instância em 2006, esse era um valor surpreendentemente baixo. Uma instância é uma máquina virtual, e, a US$ 0,10 por hora, o AWS alocava duas instâncias por núcleo em um servidor multicore. Portanto, uma unidade de computador EC2 é equivalente a um AMD Opteron de 1,0 ou 1,2 GHz ou Intel Xeon daquela era.
- *Dependência (inicial) de software open source.* A disponibilidade de software de boa qualidade que não tenha problemas de licenciamento ou custos associados a executar centenas ou milhares de servidores tornou a computação como serviço muito mais econômica para a Amazon e para seus clientes. Mais recentemente, o AWS começou a oferecer instâncias incluindo softwares comerciais de terceiros a preços mais elevados.
- *Sem garantia (inicial) de serviço.* Inicialmente, a Amazon prometeu fazer somente o melhor. O baixo custo era tão atrativo que muitos podiam viver sem uma garantia de serviço. Hoje, o AWS fornece SLAs de disponibilidade de até 99,95% em serviços como o Amazon EC2 e o Amazon S3. Além disso, o Amazon S3 foi projetado para ter durabilidade de 99,999999999% ao salvar múltiplas réplicas de cada objeto através de múltiplas localidades. Ou seja, as chances de perder permanentemente um objeto são de uma em 100 bilhões. O AWS também fornece um Painel de Saúde do Serviço, que mostra o status operacional de cada um dos serviços do AWS em tempo real, de modo que o uptime e o desempenho do AWS são totalmente transparentes.
- *Nenhum contrato necessário.* Em parte porque os custos são tão baixos, tudo o que é necessário para começar a usar o EC2 é um cartão de crédito.

A Figura 6.15 mostra o preço por hora dos muitos tipos de instâncias EC2 em 2011. Além de computação, o EC2 cobra por armazenamento de longo prazo e por tráfego de internet. (Não há custo para tráfego de internet dentro das regiões do AWS.) O Elastic Block Storage custa US$ 0,10 por GByte por mês e US$ 0,10 por milhão de requisições de E/S. O tráfego de internet custa US$ 0,10 por GByte indo para o EC2 e US$ 0,08 a US$ 0,15 por GByte deixando o EC2, dependendo do volume. Colocando isso em uma perspectiva histórica, por US$ 100 ao mês você pode usar a capacidade equivalente à soma das capacidades de todos os discos magnéticos produzidos em 1960!

Instância	Por hora	Razão para pequenos	Unidades computacionais	Núcleos virtuais	Unidades computacionais/núcleo	Memória (GB)	Disco (GB)	Tamanho do endereço
Micro	$ 0,020	0,5-2,0	0,5-2,0	1	0,5-2,0	0,6	EBS	32/64 bit
Padrão pequeno	$ 0,085	1,0	1,0	1	1,00	1,7	160	32 bit
Padrão grande	$ 0,340	4,0	4,0	2	2,00	7,5	850	64 bit
Padrão extragrande	$ 0,680	8,0	8,0	4	2,00	15,0	1.690	64 bit
Extragrande High-Memory	$ 0,500	5,9	6,5	2	3,25	17,1	420	64 bit
Extragrande duplo High-Memory	$ 1,000	11,8	13,0	4	3,25	34,2	850	64 bit
Extragrande quádruplo High-Memory	$ 2,000	23,5	26,0	8	3,25	68,4	1.690	64 bit
High-CPU médio	$ 0,170	2,0	5,0	2	2,50	1,7	350	32 bit
High-CPU extragrande	$ 0,680	8,0	20,0	8	2,50	7,0	1.690	64 bit
Extragrande quádruplo Cluster	$ 1,600	18,8	33,5	8	4,20	23,0	1.690	64 bit

FIGURA 6.15 Preço e características de instâncias EC2 sob demanda nos Estados Unidos e na região da Virgínia, em janeiro de 2011. Instâncias micro são a categoria mais nova e mais barata, e oferecem rápidas explosões de até duas unidades de computação por apenas US$ 0,02 por hora. Os clientes reportaram que as instâncias micro têm, em média, 0,5 unidade de computação. Instâncias de computação em cluster na última linha, que o AWS identifica como servidores Intel Xeon X5570 de soquete duplo com quatro núcleos por soquete rodando a 2,93 GHz, oferecem redes de 10 Gigabit/s. Eles são voltados para aplicações HPC. O AWS também oferece instâncias Spot a um custo muito mais baixo, em que você estabelece o preço que está disposto a pagar e o número de instâncias que está disposto a rodar, e então o AWS vai rodar quando o preço do Spot ficar abaixo de seu nível. Eles rodam até que você os pare ou o preço do spot exceda seu limite. Uma amostra durante o dia em janeiro de 2011 descobriu que o preço do spot era 2,3-3,1 vezes mais baixo, dependendo do tipo de instância. O AWS também oferece instâncias reservadas para casos em que os clientes sabem que vão usar a maior parte da instância por um ano. Você paga uma taxa anual por instância e uma taxa horária, que é de cerca de 30% da coluna 1, para usá-lo. Se você usou uma instância reservada 100% de um ano inteiro, o custo médio por hora, incluindo a amortização da taxa anual, será de cerca de 65% da taxa na primeira coluna. O servidor equivalente àqueles nas Figuras 6.13 e 6.14 seria uma instância padrão extragrande ou extragrande com High-CPU, para as quais calculamos o preço de US$ 0,11 por hora.

Exemplo Calcule o custo de rodar os serviços MapReduce médios na Figura 6.2, na página 385, em EC2. Considere que há serviços suficientes, então não há custos extras significativos a arredondar para obter um número de horas inteiro. Ignore os custos mensais de armazenamento, mas inclua o custo das E/S de disco para o armazenamento flexível de bloco (Elastic Block Storage — EBS) do AWS. A seguir, calcule o custo anual para rodar todos os serviços MapReduce.

Resposta A primeira questão é: qual é o tamanho correto de instância para atender ao servidor típico no Google? A Figura 6.21, na página 411, na Seção 6.7, mostra que em 2007 um servidor típico do Google tinha quatro núcleos rodando a 2,2 GHz com 8 GB de memória. Como uma única instância é um núcleo virtual equivalente a um AMD Opteron de 1-1,2 GHz, a melhor correspondência na Figura 6.15 é a extragrande com High-CPU com oito núcleos de 7,0 GB de memória. Para simplificar, vamos considerar que o acesso EBS médio ao armazenamento é de 64 KB para calcular o número de E/Ss.

A Figura 6.16 calcula o custo médio e total por ano de executar a carga de trabalho Google MapReduce no EC2. O serviço MapReduce médio de 2009 custaria pouco menos de US$ 40 no EC2, e a carga de trabalho total para 2009 custaria US$ 133 milhões no AWS. Observe que os acessos EBS correspondem a cerca de 1% dos custos totais desses serviços.

	4 Ago	6 Mar	7 Set	9 Set
Tempo médio de conclusão (horas)	0,15	0,21	0,10	0,11
Número médio de servidores por serviço	157	268	394	488
Custo por hora de EC2da instância de High-CPU XL	$ 0,68	$ 0,68	$ 0,68	$ 0,68
Custo médio de EC2 por serviço MapReduce	$ 16,35	$ 38,47	$ 25,56	$ 38,07
Número médio de requisições E/S EBS (milhões)	2,34	5,80	3,26	3,19
Custo EBS por milhão de requisições de E/S	$ 0,10	$ 0,10	$ 0,10	$ 0,10
Custo médio de E/S EBS por serviço MapReduce	$ 0,23	$ 0,58	$ 0,33	$ 0,32
Custo médio por serviço MapReduce	$ 16,58	$ 39,05	$ 25,89	$ 38,39
Número anual de serviços MapReduce	29.000	171.000	2.217.000	3.467.000
Custo total de serviços MapReduce em EC2/EBS	$ 480.910	$ 6,678,011	$ 57.394.985	$ 133.107.414

FIGURA 6.16 Custo estimado se você rodar a carga de trabalho Google MapReduce (Fig. 6.2) usando preços de 2011 para AWS ECS e EBS (Fig. 6.15).
Como estamos usando preços de 2011, essas estimativas são menos precisas para os primeiros anos do que para os mais recentes.

Exemplo Dado que os custos dos serviços MapReduce estão aumentando e já excedem US$ 100 milhões por ano, imagine que seu chefe queira que você investigue modos de reduzir os custos. Duas opções com custo potencialmente menor são as Instâncias Reservadas AWS e as Instâncias Spot AWS. Qual delas você recomendaria?

Resposta As Instâncias Reservadas AWS cobram uma taxa anual fixa mais uma taxa horária por uso. Em 2011, o custo anual para a Instância Extragrande com CPU Alta é de US$ 1.820 e a taxa horária é de US$ 0,24. Uma vez que pagamos pelas instâncias, usadas ou não, vamos supor que a utilização média das Instâncias Reservadas seja de 80%. Então, o preço médio por hora se torna:

$$\frac{\frac{\text{Preço anual}}{\text{Horas por ano}} + \text{Preço por hora}}{\text{Utilização}} = \frac{\frac{\$1820}{8760} + \$0,24}{80\%} = (0,21 + 0,24) \times 1,25 = \$0,56$$

Assim, a economia usando Instâncias Reservadas seria de aproximadamente 17%, ou US$ 23 milhões para a carga de trabalho MapReduce de 2009.

Usando alguns dias de janeiro de 2011 como amostra, o custo horário de uma Instância Spot Extragrande com High-CPU é, em média, de US$ 0,235. Como esse é o preço mínimo a oferecer para obter um servidor, não pode ser o custo médio, já que você geralmente quer executar as tarefas até o fim sem ser interrompido. Vamos considerar que você precise pagar o dobro do preço mínimo para rodar grandes serviços MapReduce até sua conclusão. A economia de custos para Instâncias Spot para a carga de trabalho de 2009 seria de cerca de 31%, ou US$ 41 milhões.

Assim, sugira Instâncias Spot a seu chefe, já que há menos compromisso inicial e elas podem economizar mais dinheiro. Entretanto, diga a ele que precisa tentar rodar serviços MapReduce em Instâncias Spot para ver o que acaba pagando para garantir que os serviços sejam executados até sua conclusão e que realmente existem centenas de Instâncias Extragrandes com High-CPU disponíveis para rodar esses serviços diariamente.

Além do baixo custo de um modelo de pagamento por uso para computação de serviços, outro ponto atraente para os usuários de computação em nuvem é que os provedores de computação em nuvem assumem os riscos do provisionamento em excesso ou insuficiente. Evitar o risco é um presente dos céus para as empresas start-up, já que qualquer erro poderia ser fatal. Se grande parte do precioso investimento for gasta nos servidores antes que o produto esteja pronto para uso pesado, a empresa poderá ficar sem dinheiro. Se o

serviço se tornar popular de repente, mas não houver servidores suficientes para atender a demanda, a empresa poderá causar uma impressão muito ruim nos novos clientes de que ela precisa desesperadamente para crescer.

O modelo para esse cenário é o FarmVille, da Zynga, um jogo do Facebook. Antes do FarmVille ser anunciado, o maior jogo social tinha cerca de cinco milhões de jogadores por dia. O FarmVille tinha um milhão de jogadores quatro dias depois do lançamento e 10 milhões depois de 60 dias. Depois de 270 dias, tinha 28 milhões de jogadores por dia e 75 milhões por mês. Como eles foram implementados no AWS, conseguiram crescer de acordo com o número de usuários. Além do mais, puderam modificar a carga de acordo com a demanda dos consumidores.

Empresas mais fortes também estão tirando proveito da escalabilidade da nuvem. Em 2011, A Netflix migrou seu site e serviço de streaming de vídeo de um datacenter convencional para o AWS. O objetivo da Netflix era permitir aos usuários assistir a um filme, por exemplo, no telefone celular, enquanto estão indo para casa, e continuar a vê-lo do ponto em que pararam na televisão de casa. Esse efeito envolve processamento de lote para converter novos filmes na variedade de formatos necessários para disponibilizá-los em telefones celulares, tablets, laptops, videogames e gravadores de vídeo digital. Esses serviços de lote do AWS podem ocupar milhares de máquinas por várias semanas para completar as conversões. O final da transação de streaming é feito no AWS e a entrega de arquivos codificados através de redes de entrega de conteúdo, como o Akamai e o Level 3. O serviço on-line sai muito menos caro do que enviar DVDs pelo correio, e o baixo custo resultante tornou o novo serviço bastante popular. Um estudo mostrou que a Netflix tem 30% do tráfego de download na internet nos Estados Unidos durante períodos de pico, à noite (em contraste, o YouTube tem somente 10% no mesmo período, entre 20 h e 22 h). De fato, a média geral é de 22% do tráfego na internet, tornando a Netflix responsável pela maior parte do tráfego da internet na América do Norte. Apesar das taxas de crescimento aceleradas das assinaturas da Netflix, a taxa de crescimento do datacenter da empresa foi interrompido e toda a capacidade de expansão está sendo feita através do AWS.

A computação em nuvem tornou os benefícios do WSC disponíveis para todos. Ela oferece associatividade de custo com a ilusão de escalabilidade infinita sem custo extra para o usuário: 1.000 servidores por uma hora custam não mais do que um servidor por 1.000 horas. Depende do provedor de computação de nuvem garantir que existam servidores, armazenamento e largura de banda suficientementes disponíveis para atender a demanda. A cadeia de fornecimento otimizada mencionada, que reduz o tempo de entrega para uma semana no caso de novos computadores, ajuda a dar essa ilusão sem levar o provedor à falência. Essa transferência de riscos, associatividade de custo e preços "pague de acordo com o uso" é um argumento poderoso para que empresas de vários tamanhos usem a computação em nuvem.

Duas questões cruzadas que modelam o custo-desempenho dos WSCs e, portanto, a computação de nuvem são a rede de WSC e a eficiência do hardware e do software de servidor.

6.6 QUESTÕES CRUZADAS

O equipamento de rede é o SUV do datacenter.

James Hamilton (2009)

A rede WSC como um gargalo

A Seção 6.4 mostrou que o equipamento de rede acima do switch de rack é uma fração significativa do custo de um WSC. Totalmente configurado, o preço de lista de um datacenter de 1 Gbit com 128 portas da Juniper (EX8216) é de US$ 716.000 sem interface óptica ou de US$ 908.000 com elas. (Esses preços sofreram um grande desconto, mas ainda

são mais de 50 vezes o preço de um sistema de rack.) Esses switches também tendem a ser grandes consumidores de energia. Por exemplo, o EX8216 consome cerca de 19.200 watts, 500-1.000 vezes mais do que um servidor em um WSC. Além do mais, esses grandes switches são configurados manualmente e frágeis em grande escala. Devido ao preço, é difícil pagar mais do que redundância dupla em um WSC usando esses grandes switches, o que limita as opções de tolerância de falha (Hamilton, 2009).

Entretanto, o impacto real sobre os switches consiste em como o superdimensionamento afeta o projeto de software e o posicionamento de serviços e dados dentro do WSC. A rede ideal de WSC seria uma caixa-preta cuja topologia e largura de banda não são interessantes, porque não há restrições: você pode executar qualquer carga de trabalho em qualquer lugar e otimizar para utilização de servidor no lugar de tráfego de rede. Os gargalos de rede de WSC de hoje restringem o posicionamento de dados, que por sua vez complica o software do WSC. Como esse software é um dos ativos mais valiosos de uma empresa de WSC, o custo extra dessa complexidade pode ser significativo.

Para os leitores interessados em aprender mais sobre o projeto de switch, o Apêndice F descreve os problemas envolvidos no projeto das redes de interconexão. Além disso, Thacker (2007) propôs pegar emprestada tecnologia de rede dos supercomputadores para superar o preço e problemas de desempenho. Vahdat *et al*. (2010) também fizeram isso e propuseram uma infraestrutura de rede que pode ser escalada para 100.000 portas e 1 petabit/s de largura de banda de bisseção. O maior benefício dessa novela de switches de datacenters é simplificar os desafios de software devidos ao oversubscrition.

Usando energia com eficiência dentro do servidor

Enquanto o PUE mede a eficiência de um WSC, não diz nada sobre o que acontece dentro do próprio equipamento de TI. Assim, outra fonte de ineficiência elétrica não mostrada na Figura 6.9 é a fonte de energia *dentro* do servidor, que converte uma entrada de 208 volts ou 110 volts nas tensões que os chips e discos usam, geralmente 3,3, 5 e 12 volts. Os 12 volts são ainda mais reduzidos, para 1,2-1,8 volts na placa, dependendo do que o microprocessador e a memória precisam. Em 2007, muitas fontes de alimentação tinham de 60-80% de eficiência; isso significa que houve maiores perdas dentro do servidor em relação ao que havia ao longo das muitas etapas e mudanças de tensão desde as linhas de alta tensão, nas torres de transmissão, para fornecer as linhas de baixa tensão no servidor. Uma razão é que elas precisavam fornecer várias tensões para os chips e para os discos, já que eles não têm ideia do que há na placa-mãe. Uma segunda razão é que, muitas vezes, a fonte de alimentação é superdimensionada em watts para o que há na placa. Além do mais, tais fontes de alimentação muitas vezes têm sua pior eficiência com carga de 25% ou menos, embora, como mostra a Figura 6.3 na página 387, muitos servidores WSC operem nessa faixa. As placas-mãe de computadores também têm módulos reguladores de tensão (Voltage Regulator Modules — VRMs), que também podem ter eficiência relativamente baixa.

Para melhorar o estado da arte, a Figura 6.17 mostra os padrões Climate Savers Computing Initiative (2007) para classificar fontes de alimentação e seus objetivos ao longo do tempo. Observe que o padrão especifica demandas de 20-50% de carga, além de 100% de carga.

Além da fonte de alimentação, Barroso e Hölzle (2007) disseram que o objetivo de todo servidor deve ser a *proporcionalidade energética*; ou seja, os servidores consumem energia em proporção à quantidade de trabalho realizado. A Figura 6.18 mostra o quanto estamos longe de atingir esse objetivo ideal usando SPECPower, um benchmark de servidor que mede a energia usada em diferentes níveis de desempenho (Cap. 1). A linha de energia proporcional foi adicionada ao uso real de energia do servidor mais eficiente para o SPECPower em julho de 2010. A maioria dos servidores não era tão eficiente. Era até 2,5

Condicionamento de carga	Base	Bronze (Junho 2008)	Prata (Junho 2009)	Ouro (Junho 2010)
20%	80%	82%	85%	87%
50%	80%	85%	88%	90%
100%	80%	82%	85%	87%

FIGURA 6.17 Classificações e objetivos de eficiência para fontes de alimentação ao longo do tempo para a Climate Savers Computing Initiative.
Essas classificações referem-se a unidades de alimentação de energia Multi-Output, que se referem a fontes de alimentação de desktop e servidor em sistemas não redundantes. Existe um padrão ligeiramente mais alto para PSUs de saída única, que são geralmente usadas em configurações redundantes (1U/2U de soquete simples, duplo ou quádruplo e servidores blade).

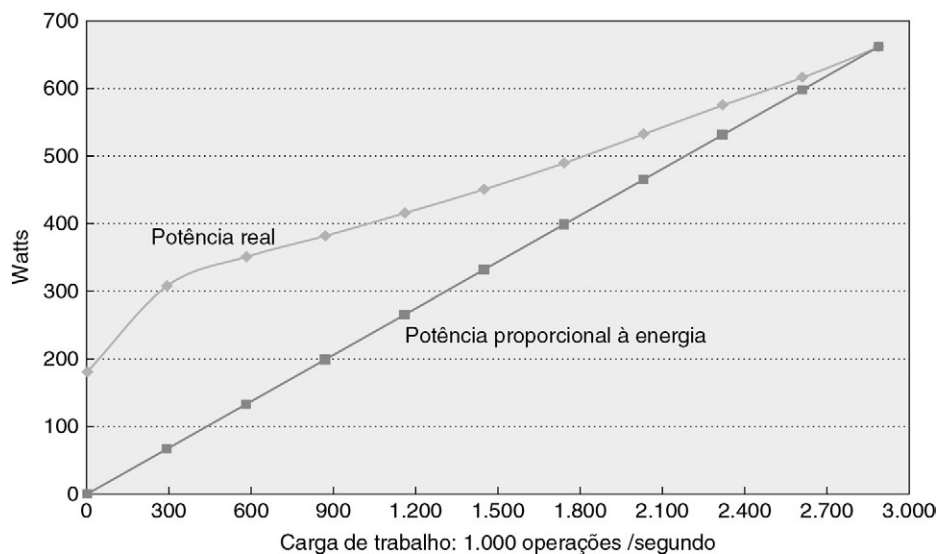



FIGURA 6.18 Os melhores resultados do SPECPower em julho de 2010, em comparação com o comportamento ideal de proporcionalidade energética.
O sistema foi o HP ProLiant SL2x170z G6, que usa um cluster de quatro Intel Xeon L5640s de soquete duplo, com cada soquete tendo seis núcleos rodando a 2,27 GHz. O sistema tinha 64 GB de DRAM e um pequeno SSD de 60 GB para armazenamento secundário. (O fato de que a memória principal é maior do que a capacidade de disco sugere que esse sistema foi preparado para esse benchmark.) O software usado foi a Máquina Virtual Java da IBM versão 9 e o Windows Server 2008, Enterprise Edition.

vezes melhor do que outros sistemas medidos naquele ano, e, no final de uma competição de benchmark, os sistemas muitas vezes são configurados de modo a vencer o benchmark, que não são típicos dos sistemas em campo. Por exemplo, os servidores com melhor classificação SPECPower usam discos de estado sólido, cuja capacidade é menor do que a da memória principal! Mesmo assim, esse sistema tão eficiente usa quase 30% da potência total quando ocioso e quase 50% com carga de somente 10%. Assim, a proporcionalidade energética permanece um objetivo elevado em vez de uma realização.

O sistema de software foi projetado para usar todos os recursos disponíveis se melhorar potencialmente o desempenho, sem preocupação com as implicações energéticas. Por exemplo, os sistemas operacionais usam toda a memória para dados de programas ou para caches de arquivo, apesar de muitos dados provavelmente nunca serem usados. Os arquitetos de software precisam considerar a energia, assim como o desempenho nos projetos futuros (Carter e Rajamani, 2010).

Exemplo Usando os dados de tipo na Figura 6.18, qual é a economia energética mudando de cinco servidores com 10% de utilização em comparação para um servidor com 50% de utilização?

Resposta Um único servidor com 10% de carga tem 308 watts e um com 50% de carga tem 451 watts. A economia então é

$$5 \times 308 / 451 = (1.540 / 451) \approx 3,4$$

ou um fator de cerca de 3,4. Se quisermos ser bons administradores ambientais no nosso WSC, devemos consolidar os servidores quando as utilizações caírem, comprar servidores que sejam mais proporcionais energeticamente ou encontrar algo que seja útil executar em períodos de baixa atividade.

Dado o background dessas seis seções, estamos prontos para apreciar o trabalho dos arquitetos do WSC Google.

6.7 JUNTANDO TUDO: O COMPUTADOR EM ESCALA WAREHOUSE DO GOOGLE

Como muitas empresas com WSCs estão concorrendo vigorosamente no mercado, até bem recentemente elas estavam relutantes em compartilhar suas inovações mais recentes com o público (e umas com as outras). Em 2009, o Google descreveu um WSC estado da arte de 2005. O Google graciosamente forneceu uma atualização do status de 2007 do seu WSC, tornando esta seção a descrição mais atualizada de um WSC do Google (Clidaras, Johnson e Felderman, 2010). Mais recentemente, o Facebook descreveu seu último datacenter como parte de http://opencompute.org.

Contêineres

Tanto o Google quanto a Microsoft construíram WSCs usando contêineres de envio. A ideia de construir um WSC a partir de contêineres é tornar o projeto dos WSCs modular. Cada contêiner é independente, e as únicas conexões externas são rede, energia e água. Os contêineres, por sua vez, fornecem rede, energia e refrigeração para os servidores em seu interior, então o trabalho do WSC é fornecer rede, energia e água fria para os contêineres e bombear a água quente resultante para torres de resfriamento e refrigeradores externos.

O WSC do Google que estamos examinando contém 45 contêineres com 40 pés de comprimento em um espaço de 300 pés por 250 pés, ou 75.000 pés quadrados (cerca de 7.000 metros quadrados). Para caber no depósito, 30 dos contêineres são colocados em pilhas de dois, ou 15 pares de contêineres empilhados. Embora a localização não tenha sido revelada, ele foi construído na época em que o Google desenvolveu WSCs em The Dales, Oregon, que proporcionam um clima moderado e ficam próximos de uma fonte de energia hidroelétrica barata e um backbone de internet. Esse WSC oferece 10 megawatts com um PUE de 1,23 nos 12 meses anteriores. Desses 0,230 de overhead de PUE, 85% vão para perdas de resfriamento (0,195 PUE) e 15% (0,035) para perdas de energia. O sistema foi ativado em novembro de 2005, e esta seção descreve seu status em 2007.

Um contêiner do Google pode suportar até 250 kilowatts. Isso significa que o contêiner pode suportar 780 watts por pé quadrado (0,09 metro quadrado), ou 133 watts por pé quadrado por todo o espaço de 75.000 pés quadrados com 40 contêineres. Entretanto, nesse WSC os contêineres têm cerca de 222 kilowats.

A Figura 6.19 é um diagrama em corte de um contêiner do Google. Um contêiner contém até 1.160 servidores, então 45 contêineres têm espaço para 52.200 servidores (esse WSC tem cerca de 40.000 servidores). Os servidores são colocados em pilhas de 20 contêineres,

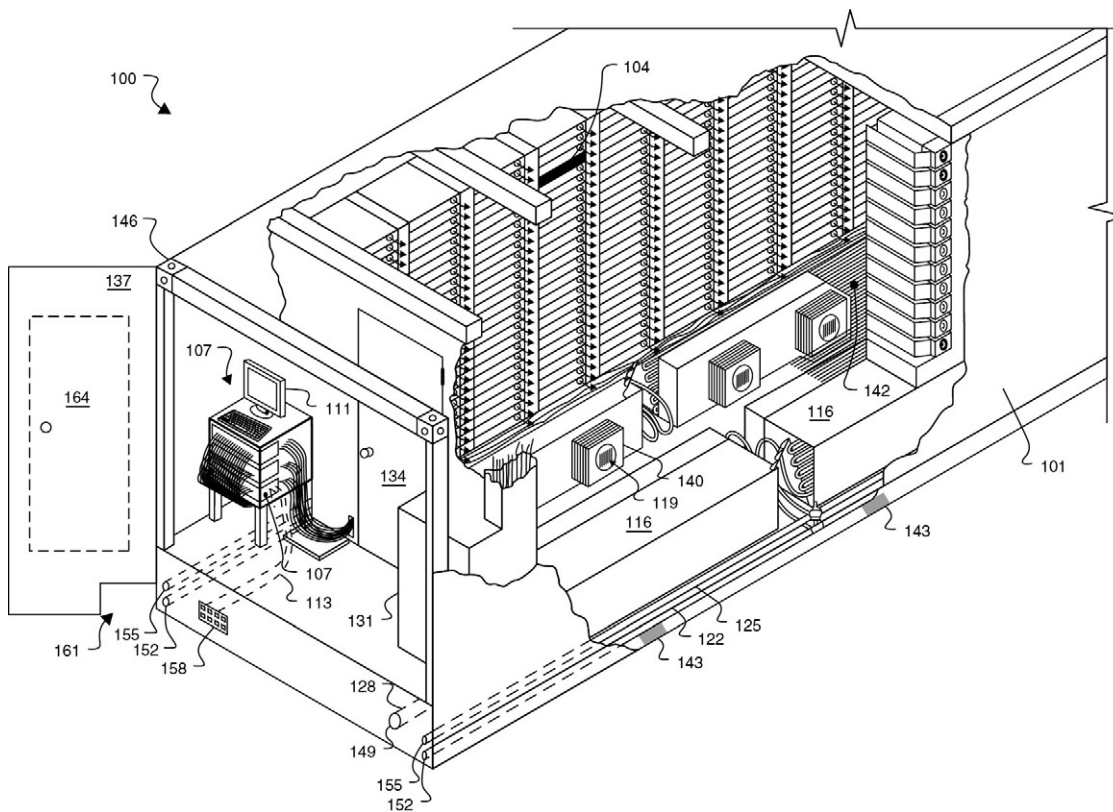



FIGURA 6.19 O Google personaliza um contêiner 1AAA padrão: 12,2 × 2,4 × 2,9 metros (40 × 8 × 9,5 pés).
Os servidores são colocados em pilhas de até 20 contêineres em racks que formam duas filas longas de 29 racks cada um, com uma linha de cada lado do contêiner. O corredor de resfriamento passa pelo centro do contêiner, e o retorno de ar quente ocorre no exterior. A estrutura de rack suspenso torna mais fácil reparar o sistema de resfriamento sem remover os servidores. Para permitir que as pessoas entrem para reparar os componentes, o contêiner contém sistemas de segurança para extinção e supressão de incêndios, saídas e iluminação de emergência e desligamento automático da energia. Os contêineres também têm muitos sensores: temperatura, pressão do fluxo de ar, detecção de vazamento de ar e iluminação detectora de movimentos. Um *tour* em vídeo do datacenter pode ser encontrado em www.google.com/corporate/green/datacenters/summit.html. A Microsoft, o Yahoo! e muitas outras estão construindo datacenters modulares com base nessas ideias, mas pararam de usar contêineres padrão ISO, uma vez que o tamanho é inconveniente.

em racks que formam duas filas longas de 29 racks (também chamados *baias*), cada qual com uma linha de cada lado do contêiner. Os switches de rack são switches de ethernet de 1 Gbit/s com 48 portas, que são posicionados em racks alternados.

Refrigeração e energia no WSC do Google

A Figura 6.20 é uma seção cruzada do contêiner que mostra o fluxo de ar. Os racks do computador são presos ao teto do contêiner. A refrigeração está abaixo de um piso elevado que desemboca no corredor entre os racks. O ar quente é retornado por detrás dos racks. O espaço restrito do contêiner impede a mistura de ar quente e ar frio, que melhora a eficiência da refrigeração. Ventoinhas de velocidade variável funcionam na menor velocidade necessária para resfriar o rack, em vez de em uma velocidade constante.

O ar "frio" é mantido a 27 °C (81 °F), o que é morno em comparação às temperaturas de muitos datacenters convencionais. Uma das razões pelas quais os datacenters geralmente rodam em ambiente tão frio não é o equipamento de TI, mas a intenção de garantir que os pontos quentes dentro do datacenter não causem problemas isolados. Ao controlar

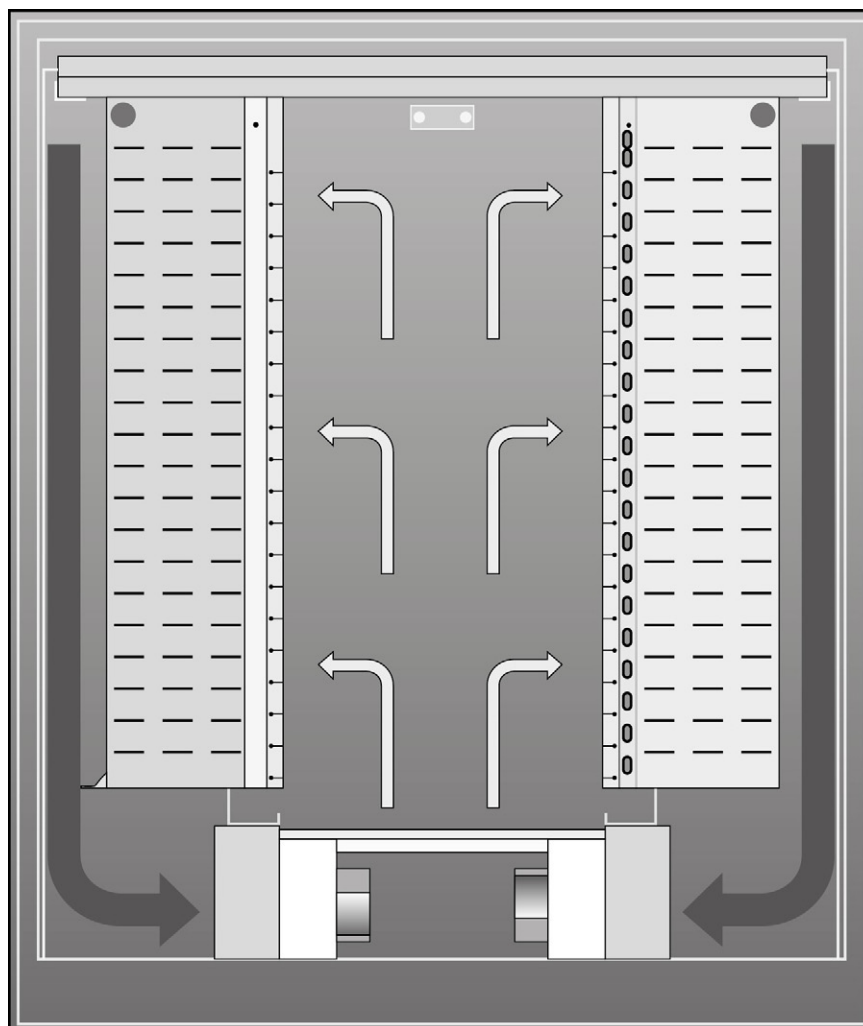



FIGURA 6.20 Fluxo de ar dentro do contêiner mostrado na Figura 6.19.
O diagrama de seção cruzada mostra dois racks de cada lado do contêiner. O ar frio sopra para o corredor no meio do contêiner e é sugado pelos servidores. O ar quente, então, retorna nas extremidades do contêiner. Esse projeto isola os fluxos de ar quente e frio.

cuidadosamente o fluxo de ar para impedir pontos quentes, o contêiner pode funcionar a uma temperatura muito maior.

Os resfriadores externos têm cortes para que, se o clima estiver correto, somente as torres de refrigeração externa precisem resfriar a água. Os resfriadores são ignorados se a temperatura da água que deixa a torre de resfriamento é de 70 °F (21 °C) ou menos.

Observe que, se estiver muito frio no exterior, as torres de resfriamento precisarão de aquecedores para evitar que se forme gelo. Uma das vantagens de localizar um WSC em The Dales é que a temperatura de bulbo úmido anual varia entre –9 °C e 19 °C (15 °F e 66 °F) com média de 5 °C (41 °F), então muitas vezes os refrigeradores podem ser desligados. Em contraste, Las Vegas, Nevada, varia de –41 °C a 17 °C (–42 °F a 62 °F), com média de –2 °C (29 °F). Além disso, tendo de resfriar para somente 27 °C (81 °F) no interior do contêiner, torna muito mais provável que a mãe natureza seja capaz de resfriar a água.

A Figura 6.21 mostra o servidor designado pelo Google para esse WSC. Para melhorar a eficiência da alimentação energética, ele fornece somente 12 volts para a placa-mãe,

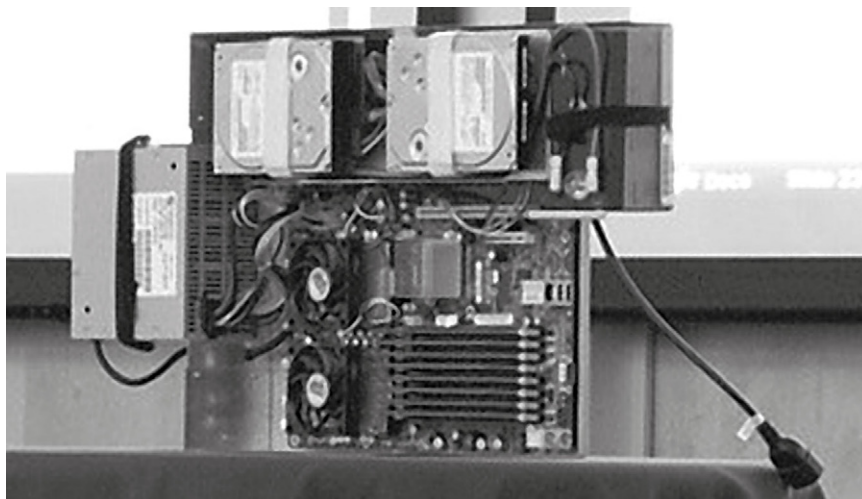



FIGURA 6.21 Servidor para o WSC do Google.
A alimentação de energia está à esquerda, e os dois discos estão no topo. As duas ventoinhas abaixo do disco esquerdo cobrem os dois soquetes do microprocessador AMD Barcelona, cada qual com dois núcleos, sendo executados a 2,2 GHz. Os oito DIMMS embaixo, à direita, contêm cada qual um 1 GB, totalizando 8 GB. Não existe chapa metálica adicional, uma vez que os servidores são ligados à bateria e há uma câmara de admissão separada no rack para cada servidor que visa ajudar a controlar o fluxo de ar. Em parte devido à altura das baterias, cabem 20 servidores em um rack.

que, por sua vez, fornece somente o suficiente para o número de discos que ele possui na placa (os laptops alimentam seus discos de modo similar). A norma do servidor é fornecer diretamente os muitos níveis de tensão necessários para os discos e chips. Essa simplificação significa que a fonte de alimentação de 2007 pode funcionar com 92% de eficiência, indo além da classificação Ouro para fontes de alimentação em 2010 (Fig. 6.17).

Os engenheiros do Google perceberam que 12 volts significavam que a UPS seria simplesmente uma bateria-padrão em cada prateleira. Portanto, em vez de ter uma sala separada para baterias, o que a Figura 6.9 mostra como 94% de eficiência, cada servidor tem sua própria bateria de chumbo-ácido, o que é uma eficiência de 99,99%. Essa "UPS distribuída" é implementada incrementalmente com cada máquina, o que significa que não há gasto de dinheiro ou energia acima da capacidade. Eles usam unidades UPS padrão de prateleira para proteger os switches de rede.

E quanto a economizar energia usando escalonamento dinâmico de voltagem-frequência (DVFS), como descrito no Capítulo 1? O DVFS não foi implementado nessa família de máquinas, uma vez que o impacto sobre a latência foi tal que isso somente era factível em regiões de atividade muito baixa para cargas de trabalho on-line, e mesmo nesses casos a economia em todo o sistema foi muito pequena. Portanto, o complexo loop de controle administrativo necessário para implementá-lo não podia ser justificado.

Uma das chaves para alcançar o PUE de 1,23 foi instalar dispositivos de medição (chamados *transformadores de corrente*) em todos os circuitos através dos contêineres e em todos os outros pontos do WSC para medir o uso real de energia. Essas medições permitiram ao Google ajustar o projeto do WSC ao longo do tempo.

O Google publica o PUE do seus WSC a cada trimestre. A Figura 6.22 mostra o PUE de 10 WSCs do Google do terceiro trimestre de 2007 até o segundo trimestre de 2010. Esta seção descreve o WSC chamado Google A. O Google E opera com um PUE de 1,16 com a refrigeração sendo de somente 0,105, devido às maiores temperaturas operacionais e

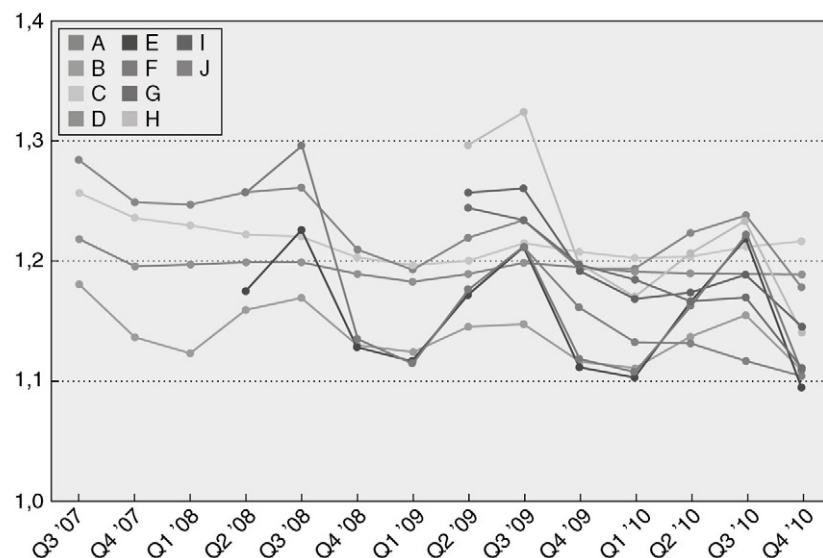



FIGURA 6.22 **Eficiência de uso de energia (PUE) de 10 WSCs Google ao longo do tempo.**
O WSC descrito nesta seção é o Google A. Ele é a linha mais alta em Q3'07 e Q2'10 (de www.google.com/corporate/green/datacenters/measuring.htm). Recentemente, o Facebook anunciou um novo datacenter que deve entregar um impressionante PUE de 1,07 (http://opencompute.org/). A instalação de Prineville, Oregon, não possui ar-condicionado e água refrigerada. Ela depende estritamente do ar externo, trazido por um lado do edifício, filtrado, resfriado através de nebulizadores, bombeado através do equipamento de TI e então enviado para fora do edifício por ventiladores de exaustão. Além disso, os servidores usam uma fonte de alimentação personalizada que permite ao sistema de distribuição de energia pular uma das etapas de conversão de tensão mostradas na Figura 6.9.

cortes de resfriadores. A distribuição de energia é de somente 0,039, devido aos UPS distribuídos e à fonte de alimentação de tensão única. O melhor resultado do WSC foi 1,12, com o Google A a 1,23. Em abril de 2009, a média de 12 meses ponderada por uso de todos os datacenters foi de 1,19.

Servidores em um WSC Google

O servidor mostrado na Figura 6.21 tem dois soquetes, cada qual contendo um processador AMD Opteron com dois núcleos rodando a 2,2 GHz. A foto mostra oito DIMMs, e esses servidores são tipicamente implementadas com 8 GB de DRAM DDR2. Um novo recurso é que o barramento de memória tem seu clock reduzido para 533 MHz em relação aos 666 MHz padrão, uma vez que o barramento mais lento tem pouco impacto no desempenho e significativo impacto no consumo de energia.

O projeto básico tem um único cartão de interface de rede (Network Interface Card — NIC) para um link ethernet de 1 Gbit/s. Embora a Figura 6.21 mostre dois drives com discos SATA, o servidor básico tem somente um. O pico de potência da base é de cerca de 160 watts, e a potência ociosa é de 85 watts.

Esse nó básico é suplementado para oferecer um nó de armazenamento (ou "diskfull"). Primeiro, uma segunda bandeja contendo 10 discos SATA é conectada ao servidor. Para ter mais um disco, um segundo disco é colocado no local vazio na placa-mãe, dando ao nó de armazenamento 12 discos SATA. Finalmente, uma vez que um nó de armazenamento poderia saturar um único link ethernet de 1Gbit/s, um segundo NIC ethernet foi adicionado. O pico de potência de um nó de armazenamento é de cerca de 300 watts, com 198 watts quando ocioso.

Observe que o nó de armazenamento ocupa dois slots no rack, que é uma das razões pelas quais o Google implementou 40.000 em vez de 52.2000 servidores nos 45 contêineres. Nessa instalação, a razão foi de cerca de dois nós computacionais para cada nó de armazenamento, mas essa razão varia muito entre os WSCs do Google. Portanto, o Google A tinha cerca de 190.000 discos em 2007, ou uma média de quase cinco discos por servidor.

Rede em um WSC Google

Os 40.000 servidores estão divididos em três arrays de mais de 10.000 servidores cada um (os arrays são chamados *clusters* na terminologia do Google). O switch de rack de 48 portas usa 40 portas para se conectar aos servidores, deixando oito para uplinks para os switches de array.

Os switches de array são configurados para suportar até 480 links ethernet de 1 Gbit/s e algumas portas de 10 Gbit/s. As portas de 1 Gigabit são usadas para conectar com os switches de rack, já que cada switch de rack tem um único link para cada um dos switches de array. As portas de 10 Gbit/s se conectam a cada um de dois roteadores do datacenter, que agregam todos os roteadores de array e fornecem conectividade com o mundo exterior. O WSC usa dois roteadores de datacenter para confiabilidade, então uma única falha de roteador de datacenter não ocupa todo o WSC.

O número de portas de uplink usadas por switch de rack varia de um mínimo de duas a um máximo de oito. No caso das portas duplas, os switches de rack operam com uma taxa de oversubscripition de 20:1. Ou seja, há 20 vezes a largura de banda de rede dentro do switch em relação ao que sai dele. As aplicações com demandas de tráfego significativas além de um rack tendem a sofrer com o pobre desempenho da rede. Portanto, o projeto de uplink de oito portas, que fornecia uma taxa de oversubscripition menor, de apenas 5:1, foi usado para arrays com requisitos de tráfego mais exigentes.

Monitoramento e reparo de um WSC do Google

Para que um único operador seja responsável por mais de 1.000 servidores, você precisa de uma infraestrutura de monitoramento extensa e de alguma automação para ajudar nos eventos de rotina.

O Google implementa software de monitoramento para rastrear a saúde de todos os servidores e equipamentos de rede. Os diagnósticos estão sendo executado o tempo todo. Quando um sistema falha, muitos dos problemas possíveis têm soluções automatizadas simples. Nesse caso, o próximo passo é reiniciar o sistema e tentar reinstalar componentes de software. Assim, o procedimento trata da maioria das falhas.

As máquinas que falham nessas primeiras etapas são adicionadas a uma fila de máquinas a serem reparadas. O diagnóstico do problema é colocado na fila juntamente com o ID da máquina que falhou.

Para amortizar o custo do reparo, máquinas que falharam são endereçadas em lotes por técnicos de reparo. Quando o software de diagnóstico faz uma avaliação com alto grau de confiança, a parte é imediatamente substituída sem passar pelo processo de diagnóstico manual. Por exemplo, se o diagnóstico disser que o disco 3 de um nó de armazenamento está estragado, ele será substituído imediatamente. Máquinas que falharam sem diagnóstico ou com diagnósticos com baixo grau de confiabilidade são examinadas manualmente.

O objetivo é ter menos de 1% de todos os nós na fila de reparo manual a qualquer dado momento. O tempo médio na fila de reparo é de uma semana, embora leve muito menos tempo para o técnico reparar o equipamento. A maior latência sugere a importância do

throughput de reparo, o que afeta o custo de operações. Observe que os reparos automatizados da primeira etapa levam desde alguns minutos para fazer uma reinicialização/reinstalação até horas para realizar testes direcionados de estresse para garantir que a máquina está de fato operacional.

Essas latências não levam em conta o tempo para desativar os servidores quebrados. A razão é que uma grande variável é o conteúdo do nó. Um nó sem conteúdo pode levar muito menos tempo do que um nó de armazenamento cujos dados precisem ser evacuados antes que ele possa ser substituído.

Resumo

Em 2007, o Google já havia demonstrado diversas inovações para melhorar a eficiência energética dos seus WSCs para entregar um PUE de 1,23 no Google A:

- Além de fornecer um recipiente barato para abrigar os servidores, os contêineres de carga modificados separam as câmaras de admissão de ar quente e frio, o que ajuda a reduzir a variação na temperatura do ar de entrada para os servidores. No pior caso, menos grave, o ar frio pode ser entregue a temperaturas mais quentes.
- Esses contêineres também diminuem a distância do loop de circulação de ar, o que reduz a energia para mover o ar.
- Operar os servidores a temperaturas maiores significa que o ar precisa ser resfriado a somente 27 °C (81 °F) em vez dos tradicionais 18-22 °C (64-71 °F).
- Uma temperatura-alvo maior para o ar frio ajuda a colocar a instalação com mais frequência dentro de uma faixa que pode ser sustentada por soluções de refrigeração evaporativas (torres de resfriamento), que são mais eficientes do que os resfriadores tradicionais em termos de energia.
- Implementar WSCs em climas temperados para permitir o uso de resfriamento exclusivamente evaporativo em alguns períodos do ano.
- Implementar hardware e software de monitoramento para medir o PUE real em comparação com o PUE projetado melhora a eficiência operacional.
- Operar mais servidores do que o cenário de pior caso para o sistema de distribuição de energia, sugeriria, já que é estatisticamente improvável que milhares de servidores ficassem muito ocupados simultaneamente, e ainda depender do sistema de monitoramento para reduzir o trabalho em casos improváveis de acontecer (Fan, Webber e Barroso, 2007; Ranganathan *et al*., 2006). O PUE melhora porque a instalação está operando mais próximo de sua capacidade totalmente projetada, em que ela é mais eficiente porque os servidores e sistemas de refrigeração são proporcionalmente energéticos. O aumento da utilização reduz a demanda por novos servidores e novos WSCs.
- Projetar placas-mães que só precisam de uma fonte de 12 volts, de modo que a função UPS poderia ser fornecida por baterias-padrão associadas a cada servidor em vez de uma sala com baterias, reduzindo assim os custos e reduzindo uma fonte de ineficiência na distribuição de energia dentro de um WSC.
- Projetar cuidadosamente a placa do servidor para melhorar sua eficiência energética. Por exemplo, reduzir o clock do barramento frontal nesses microprocessadores reduz o uso de energia com impacto irrelevante sobre o desempenho. (Observe que tais otimizações não impactam o PUE, mas reduzem o consumo geral de energia do WSC.)

O projeto do WSC deve ter melhorado nos anos seguintes, já que o melhor WSC do Google reduziu o PUE de 1,23 do Google A para 1,12. O Facebook anunciou em 2011 que havia reduzido o PUE para 1,07 em seu novo datacenter (http://opencompute.org/). Será interessante ver que inovações continuam a melhorar a eficiência dos WSCs de modo que sejamos

bons preservadores do nosso meio ambiente. Talvez no futuro consideremos até o custo energético para *manufaturar* os equipamentos dentro de um WSC (Chang *et al.*, 2010).

6.8 FALÁCIAS E ARMADILHAS

Apesar de os WSCs terem menos de uma década de existência, arquitetos de WSC, como os do Google, já descobriram muitas armadilhas e falácias sobre os WSCs, muitas vezes aprendidas do jeito mais difícil. Como dissemos na introdução, os arquitetos de WSC são os Saymour Crays de hoje.

Falácia. *Os provedores de computação em nuvem estão perdendo dinheiro.*

Uma questão popular sobre a computação em nuvem é se ela é lucrativa com esses baixos preços.

Com base nos preços do AWS da Figura 6.15, poderíamos cobrar US$ 0,68 por hora por servidor por computação. (O preço de US$ 0,085 por hora é para uma máquina virtual equivalente a uma unidade computacional EC2, não um servidor completo.) Se pudéssemos vender 50% das horas de servidor, isso geraria US$ 0,34 de receita por hora por servidor. (Observe que os clientes pagam, não importando quão pouco usem os servidores que ocupam; por isso, vender 50% das horas de servidor não quer dizer necessariamente que a utilização dos servidores seja de 50%).

Outro modo de calcular a receita seria usar as *Instâncias Reservadas* do AWS, em que os clientes pagam uma taxa anual para reservar uma instância e depois uma taxa menor por hora para usá-la. Combinando essas taxas, o AWS receberia US$ 0,45 de receita por hora por servidor por todo um ano.

Se pudéssemos vender 750 GB por servidor para armazenamento usando os preços do AWS, além da receita de computação, isso geraria outros US$ 75 por mês por servidor ou outros US$ 0,10 por hora.

Esses números sugerem uma receita média de US$ 0,44 por hora por servidor (através das Instâncias sob Demanda) a US$ 0,55 por hora (através de Instâncias Reservadas). Com base na Figura 6.13, nós calculamos o custo por servidor como US$ 0,11 por hora para o WSC na Seção 6.4. Embora os custos apresentados na Figura 6.13 sejam estimativas que *não* foram baseadas nos custos reais do AWS, e as vendas de 50% para processamento de servidor e 750 GB de utilização de armazenamento por servidor sejam só exemplos, essas suposições sugerem uma margem bruta de 75-80%. Considerando que esses cálculos sejam razoáveis, eles sugerem que a computação em nuvem é lucrativa, especialmente para um negócio de serviços.

Falácia. *Custos capitais da instalação WSC são maiores que os dos servidores que ela abriga.*

Embora uma rápida olhada na Figura 6.13, na página 398, possa levar você a essa conclusão, essa rápida olhada ignora o tempo de amortização de cada parte do WSC completo. Entretanto, a instalação dura 10-15 anos, enquanto os servidores precisam ser substituídos a cada 3-4 anos. Usando os tempos de amortização da Figura 6.13, de 10 anos e três anos, respectivamente, as despesas capitais ao longo de uma década foram de US$ 72 milhões para a instalação, e de 3,3 × US$ 67 milhões, ou US$ 221 milhões para os servidores. Assim, os custos capitais para servidores em um WSC ao longo de uma década são fatores três vezes maiores para a instalação do WSC.

Armadilha. *Tentar economizar energia com modos inativos de baixa energia* versus *modos ativos de baixa energia.*

A Figura 6.3, na página 387, mostra que a utilização média dos servidores está entre 10-50%. Dada a preocupação com os custos operacionais de um WSC abordados na Seção 6.4, você pode pensar que os modos de baixa energia seriam uma grande ajuda.

Como mencionamos no Capítulo 1, você não pode acessar DRAMs ou discos nesses *modos inativos de baixa energia*, então deve retornar ao modo totalmente ativo para leitura ou escrita, não importa quão baixa seja a taxa. A armadilha é que o tempo e a energia necessários para retornar a um modo totalmente ativo tornam os modos de baixa energia menos atraentes. A Figura 6.3 mostra que quase todos os servidores têm, em média, 10% de utilização, então você poderia esperar longos períodos de baixa atividade, mas não longos períodos de inatividade.

Em contraste, os processadores ainda rodam em modos de baixa potência a um pequeno múltiplo da taxa regular, de modo que os *modos ativos de baixa potência* são muito mais fáceis de usar. Observe que o tempo para os processadores mudarem para o modo totalmente ativo também é medido em microssegundos, então os modos ativos de baixa energia também endereçam os problemas de latência dos modos de baixa energia.

Armadilha. *Usar um processador muito fraco ao tentar melhorar o custo-desempenho do WSC.*

A lei de Amdahl ainda se aplica aos WSCs, já que haverá algum trabalho serial para cada requisição e que pode aumentar a latência de requisição se ela rodar em um servidor lento (Hölzle, 2010; Lim *et al*., 2008). Se o trabalho serial aumentar a latência, então o custo de usar um processador fraco deverá incluir os custos de desenvolvimento de software para otimizar o código a fim de retorná-lo para a menor latência. O maior número de threads de muitos servidores lentos também pode tornar mais difícil escalonar e equilibrar a carga; assim, a variabilidade no desempenho dos threads pode levar a maiores latências. Uma chance de uma em 1.000 de mau escalonamento provavelmente não é um problema com 10 tarefas, mas o é com 1.000 tarefas quando você tem que esperar pela tarefa mais lenta. Muitos servidores melhores também podem levar a menor utilização, já que é obviamente mais fácil escaloná-los quando há menos coisas para escalonar. Por fim, até mesmo alguns algoritmos paralelos ficam menos eficientes quando o problema é muito particionado. Atualmente, a regra de ouro do Google usa a faixa inferior de computadores classe servidor (Barroso e Hölzle, 2009).

Como exemplo concreto, Reddi *et al*. (2010) compararam microprocessadores embarcados (Atom) e microprocessadores de servidor (Nehalem Xeon) executando o mecanismo de busca Bing. Eles descobriram que a latência de uma busca foi cerca de três vezes maior no Atom do que no Xeon. Além do mais, o Xeon era mais robusto. Conforme a carga aumenta no Xeon, a qualidade do serviço degrada gradual e modestamente. O Atom rapidamente viola seu objetivo de qualidade de serviço conforme tenta absorver carga adicional.

Esse comportamento se traduz diretamente na qualidade da busca. Dada a importância da latência para o usuário, conforme sugerido na Figura 6.12, o sistema de busca Bing usa várias estratégias para refinar os resultados de busca quando a latência da busca ainda não excedeu uma latência-limite. A menor latência dos nós maiores do Xeon significa que eles podem passar mais tempo refinando os resultados da busca. Portanto, mesmo quando o Atom não tinha quase nenhuma carga, deu respostas piores do que as do Xeon em 1% das pesquisas. Em cargas normais, 2% das respostas foram piores.

Falácia. *Dadas as melhorias na confiabilidade das DRAMs e a tolerância a falhas dos sistemas de software dos WSCs, você não precisa gastar mais com memória ECC em um WSC.*

Como o ECC adiciona 8 bits para cada 64 bits de DRAM, você poderia poupar 1/9 dos custos de DRAM eliminando o código de correção de erro (ECC), especialmente uma vez

que medições da DRAM haviam afirmado taxas de falha de 1.000-5.000 FIT (falhas por milhões de horas de operação) por megabit (Tezzaron Semiconductor, 2004).

Schroeder, Pinheiro e Weber (2009) estudaram as medições das DRAMS com proteção ECC na maioria dos WSCs do Google, que com certeza possuíam centenas de milhares de servidores, durante um período de dois anos e meio. Eles descobriram taxas FIT 15-25 vezes maiores do que haviam sido publicadas, ou 25.000-70.000 falhas por megabit. As falhas afetaram mais de 8% das DIMMs, e a DIMM média tinha 4.000 erros corrigíveis e 0,2 erro não corrigíveis por ano. Medidas no servidor, cerca de um terço, experimentou erros de DRAM a cada ano, com uma média de 22.000 erros corrigíveis e um erro não corrigível por ano. Ou seja, para um terço dos servidores, um erro de memória é corrigido a cada 2,5 horas. Observe que esses sistemas usaram os códigos chipkill mais poderosos em vez dos códigos SECDED mais simples. Se o esquema mais simples tivesse sido usado, as taxas de erro não corrigíveis teriam sido 4-10 vezes maiores.

Em um WSC que tem somente proteção de erro de paridade, os servidores precisariam ser reinicializados para cada erro de paridade de memória. Se o tempo de reinicialização fosse de cinco minutos, um terço das máquinas passaria 20% do tempo sendo reinicializadas. Tal comportamento reduziria o desempenho da instalação de US$ 150 milhões em cerca de 6%. Além do mais, esses sistemas apresentariam muitos erros não corrigíveis sem que os operadores fossem notificados de sua ocorrência.

Nos primeiros anos, o Google usava uma DRAM que não tinha nem mesmo proteção de paridade. Em 2000, durante testes antes do envio da próxima versão do índice de busca, ele começou a sugerir documentos aleatórios em resposta a testes de busca (Barroso e Hölzle, 2009). A razão foi uma falha "*stuck-at-zero*" em algumas DRAMs, que corromperam o novo índice. O Google adicionou verificações de consistência para detectar tais erros no futuro. Conforme o WSC aumentou de tamanho e as DIMMs ECC ficaram mais baratas, o ECC tornou-se o padrão nos WSCs do Google. O ECC tem a vantagem adicional de tornar muito mais fácil encontrar DIMM quebradas durante reparos.

Tais dados sugerem por que a GPU Fermi (Cap. 4) adiciona ECC para essa memória onde seus predecessores não tinham nem mesmo proteção de paridade. Além do mais, essas taxas FIT criam dúvidas sobre os esforços relacionados a usar o processador Intel Atom em um WSC — devido à sua eficiência energética melhorada —, uma vez que o chip de 2011 não suporta DRAM ECC.

Falácia. *Desligar o hardware durante períodos de baixa atividade melhora o custo-desempenho de um WSC.*

A Figura 6.14, na página 399, mostra que o custo de amortizar a distribuição energética e a infraestrutura de refrigeração é 50% maior do que a conta mensal de eletricidade. Portanto, embora isso certamente fosse economizar algum dinheiro por compactar as cargas de trabalho e desligar máquinas ociosas, mesmo que você pudesse economizar metade da energia, reduziria a conta operacional somente em 7%. Haveria também problemas práticos a superar, uma vez que a extensa infraestrutura de monitoramento do WSC depende de ser capaz de "cutucar" o equipamento e ver se ele responde. Outra vantagem da proporcionalidade energética e dos modos ativos de baixa energia é que eles são compatíveis com a infraestrutura de monitoramento dos WSCs, o que permite que um único operador seja responsável por mais de 1.000 servidores.

A sabedoria convencional dos WSCs é executar outras tarefas valiosas durante períodos de baixa atividade de modo a recuperar o investimento em distribuição energética e refrigeração. Um grande exemplo são os serviços de lote MapReduce, que criam índices

de busca. Outro exemplo de agregar valor com a baixa utilização são os preços spot no AWS, que a legenda na Figura 6.15, na página 403, descreve. Os usuários do AWS que são flexíveis sobre quando suas tarefas são executadas podem poupar um fator de 2,7-3 de computação ao permitir que o AWS escalone as tarefas com mais flexibilidade usando Instâncias Spot, como quando o WSC teria outra forma de baixa utilização.

Falácia. *Substituir todos os discos com memórias flash vai melhorar o custo-desempenho de um WSC.*

A memória flash é muito mais rápida do que o disco para algumas cargas de trabalho do WSC, assim como aquelas que realizam muitas leituras e escritas aleatórias. Por exemplo, o Facebook implementou memória flash como discos em estado sólido (SSDs) como cache write-back, chamado cache-flash, como parte de seu sistema de arquivos no seu WSC, de modo que os arquivos ativos permanecessem na memória flash e os arquivos inativos permanecessem em disco. Entretanto, como todas as melhoras de desempenho em um WSC devem ser julgadas com base no custo-desempenho, antes de substituir todos os discos com SSD, a questão é realmente E/S por segundo por dólar e capacidade de armazenamento por dólar. Como vimos no Capítulo 2, a memória flash custa pelo menos 20 vezes mais por GByte do que os discos magnéticos: US$ 2,00/GByte *versus* US$ 0,09/Gbyte.

Narayanan *et al*. (2009) examinaram a migração de cargas de trabalho de disco para SSD simulando traços de carga de trabalho de pequenos e grandes datacenters. Sua conclusão foi de que os SSDs não eram eficientes em termos de custo para qualquer uma das suas cargas de trabalho, devido à baixa capacidade de armazenamento por dólar. Para atingir o ponto de break-even, os dispositivos de armazenamento com memória flash precisam melhorar a capacidade por dólar de um fator de 3-3.000, dependendo da carga de trabalho.

Mesmo quando você considera o consumo de energia na equação, é difícil justificar a substituição de discos com memória flash para dados que são acessados com pouca frequência. Um disco de 1 terabyte usa cerca de 10 watts de potência, então, usando a regra geral de US$ 2,00 por watt-ano mostrada na Seção 6.4, o máximo que você poderia economizar com a energia reduzida seria US$ 20 por ano por disco. Entretanto, o custo CAPEX em 2011 para um terabyte de armazenamento é de US$ 2.000 para memória flash e somente de US$ 90 para disco.

6.9 COMENTÁRIOS FINAIS

Herdando o título de construtores dos maiores computadores do mundo, os arquitetos de computadores WSCs estão projetando a última parte do futuro da TI que completa o cliente "móvel". Muitos de nós usamos WSCs muitas vezes por dia, e o número de vezes por dia e o número de pessoas usando WSCs certamente vão aumentar na próxima década. Mais da metade dos quase sete bilhões de pessoas no planeta já tem telefones celulares. Conforme esses dispositivos se tornem "prontos" para a internet, muito mais pessoas ao redor do mundo serão capazes de se beneficiar dos WSCs.

Além do mais, as economias de escala descobertas pelos WSCs realizaram o tão sonhado objetivo da computação como serviço. Computação em nuvem significa que qualquer um, em qualquer lugar, com boas ideias e bons modelos de negócios pode utilizar milhares de servidores para compartilhar sua visão quase instantaneamente. Sem dúvida, existem obstáculos importantes que limitariam o crescimento da computação de nuvem no tocante a padrões, privacidade e a taxa de crescimento da largura de banda da internet, mas prevemos que eles serão administrados de modo que a computação em nuvem possa florescer.

Dado o crescente número de núcleos por chip (Cap. 5), os clusters vão aumentar para incluir milhares de núcleos. Nós acreditamos que as tecnologias desenvolvidas para rodar WSCs vão se provar úteis e ser usadas nos clusters, de modo que estes vão rodar as mesmas máquinas virtuais e sistemas de software desenvolvidos para WSCs. Uma vantagem seria o suporte fácil para datacenters "híbridos", em que a carga de trabalho seria facilmente enviada à nuvem numa situação de aperto e então voltaria a depender somente da computação local.

Entre os muitos recursos atraentes da computação em nuvem, ela oferece incentivos econômicos para a conservação. Enquanto é difícil convencer os *provedores* de computação em nuvem a desligar equipamentos não utilizados para poupar energia, dado o custo do investimento em infraestrutura, é fácil convencer os *usuários* de computação em nuvem a desistir de instâncias ociosas, uma vez que eles estão pagando por elas, estejam utilizando-as de forma útil ou não. De modo similar, cobrar pelo uso encoraja os programadores a usarem computação, comunicação e armazenamento eficiente, o que pode ser difícil de encorajar sem um esquema de preços compreensível. Os preços explícitos também permitem aos pesquisadores avaliar as inovações em custo-desempenho em vez de apenas o desempenho, já que os custos são agora facilmente mensuráveis e confiáveis. Finalmente, computação em nuvem significa que os pesquisadores podem avaliar suas ideias na escala de milhares de computadores, algo pelo que, no passado, somente grandes empresas podiam pagar.

Nós acreditamos que os WSCs estão mudando os objetivos e princípios do projeto de servidores, assim como as necessidades dos clientes "móveis" estão mudando os objetivos e princípios do projeto de microprocessadores. Os dois estão revolucionando também a indústria de software. O desempenho por dólar e o desempenho por joule orientam o hardware dos clientes móveis e o hardware dos WSCs, e o paralelismo é a chave para cumprir esses grupos de objetivos.

Os arquitetos terão um papel vital nas duas metades desse mundo excitante. Estamos ansiosos para ver — e usar — o que está por vir.

6.10 PERSPECTIVAS HISTÓRICAS E REFERÊNCIAS

A Seção L.8 (disponível on-line) cobre o desenvolvimento de clusters que foram a base do WSC e da computação como serviço. (Os leitores interessados em aprender mais devem começar com Barroso e Hölzle [2009] e os posts de blog e palestras de James Hamilton em http://perspectives.mvdirona.com).

ESTUDOS DE CASO E EXERCÍCIOS POR PARTHASARATHY RANGANATHAN

Estudo de caso 1: Custo total da propriedade influenciando decisões sobre computadores em escala warehouse

Conceitos ilustrados por este estudo de caso

- Custo total da propriedade (Total Cost of Ownership — TCO)
- Influência do custo do servidor e energia sobre todo o WSC
- Benefícios e desvantagens dos servidores de baixa potência

O custo total da propriedade é uma medida importante para avaliar a efetividade de um computador em escala warehouse (WSC). O TCO inclui o CAPEX e o OPEX, descritos na Seção 6.4, e reflete o custo de propriedade de todo o datacenter para atingir certo nível de desempenho. Muitas vezes, ao se considerarem diferentes servidores, redes e

arquiteturas de armazenamento, o TCO é a medida de comparação importante usada pelos proprietários de datacenters para decidir quais são as melhores opções. Entretanto, o TCO é um cálculo multidimensional que leva em conta muitos fatores diferentes. O objetivo deste estudo de caso é fazer um exame detalhado dos WSCs, como diferentes arquiteturas influenciam o TCO e como o TCO orienta as decisões do operador. Nesse estudo de caso vamos usar os números da Figura 6.13 e da Seção 6.4, e supor que o WSC descrito atingirá o nível de desempenho desejado pelo operador. Muitas vezes, o TCO é usado para comparar diferentes opções de servidor que tenham múltiplas dimensões. Os exercícios neste estudo de caso examinam como tais comparações são feitas no contexto dos WSCs e a complexidade envolvida na tomada das decisões.

6.1 [5/5/10] <6.2, 6.4> Neste capítulo, o paralelismo em nível de dados foi discutido como um modo de os WSCs atingirem alto desempenho em grandes problemas. É concebível que um desempenho ainda maior possa ser obtido pelo uso de servidores de alto nível. Entretanto, servidores com maior desempenho muitas vezes vêm com aumento de preço não linear.

a. [5] <6.4> Supondo que os servidores sejam 10% mais rápidos na mesma utilização, mas 20% mais caros, qual é o CAPEX para o WSC?

b. [5] <6.4> Se esses servidores também usarem 15% mais energia, qual será o OPEX?

c. [10] <6.2, 6.4> Dado o aumento de velocidade e o aumento de consumo de energia, qual deverá ser o custo dos novos servidores para que eles sejam comparáveis ao cluster original? (*Dica*: Com base nesse modelo de TCO, você pode precisar mudar a carga crítica da instalação.)

6.2 [5/10] <6.4, 6.8> Para atingir um OPEX menor, uma alternativa atraente é usar versões de baixa potência dos servidores para reduzir a eletricidade total necessária para rodar os servidores. Entretanto, versões de baixa potência de componentes de alto nível similares aos servidores de alto nível também têm trade-offs não lineares.

a. [5] <6.4, 6.8> Se as opções de servidor de baixa potência oferecerem 15% menos de consumo de energia com o mesmo desempenho, mas forem 20% mais caros, serão uma boa troca?

b. [10] <6.4, 6.8> A que custo os servidores se tornam comparáveis ao cluster original? E se o preço da eletricidade dobrar?

6.3 [5/10/15] <6.4, 6.6> Servidores que têm diferentes modos de operação oferecem oportunidades para rodar dinamicamente diferentes configurações no cluster para igualar o uso de carga de trabalho. Use as informações apresentadas na Figura 6.23 para os modos de consumo de energia/desempenho para um dado servidor de baixa potência.

a. [5] <6.4, 6.6> Se um operador de servidor decidisse economizar nos custos com energia rodando todos os servidores com desempenho médio, quantos servidores seriam necessários para atingir o mesmo nível de desempenho?

b. [10] <6.4, 6.6> Quais são o CAPEX e o OPEX de tal configuração?

c. [15] <6.4, 6.6> Considerando que houvesse a alternativa de comprar um servidor 20% mais barato, porém que fosse mais lento e que usasse menos

Modo	Desempenho	Potência
Alto	100%	100%
Médio	75%	60%
Baixo	59%	38%

FIGURA 6.23 Modos de consumo de energia-desempenho para servidores de baixa potência.

energia, descubra a curva desempenho-consumo de energia que proporciona um TCO comparável ao servidor básico.

6.4 [Discussão] <6.4> Discuta os trade-offs e benefícios das duas opções dadas no Exercício 6.3 supondo que uma carga de trabalho constante seja executada nos servidores.

6.5 [Discussão] <6.2, 6.4> Ao contrário dos clusters de computação de alto desempenho (High-Performance Computing — HPC), muitas vezes os WSCs experimentam flutuação significativa na carga de trabalho ao longo do dia. Discuta os trade-offs e benefícios das duas opções dadas, desta vez supondo uma carga de trabalho que varie.

6.6 [Discussão] <6.4, 6.7> O modelo de TCO apresentado até agora ignora um número significativo de detalhes. Discuta o impacto dessas abstrações para a precisão geral do modelo TCO. Quando é seguro fazer tais abstrações? Em que casos mais detalhes proporcionariam respostas significativamente diferentes?

Estudo de caso 2: Alocação de recursos em WSCs e TCO

Conceitos ilustrados por este estudo de caso

- Provisionamento de servidor e energia em um WSC
- Variância das cargas de trabalho no tempo
- Efeitos da variância no TCO

Alguns dos principais desafios para a implementação de WSCs eficientes são provisionar corretamente recursos e utilizá-los ao máximo. Esse problema é complexo, devido ao tamanho dos WSCs, além da variância potencial das cargas de trabalho sendo executadas. Os exercícios neste estudo de caso mostram como usos diferentes de recursos podem afetar o TCO.

6.7 [5/5/10] <6.4> Um dos desafios no provisionamento em um WSC é determinar a carga energética correta, dado o tamanho da instalação. Como descrito no capítulo, muitas vezes a potência anunciada é um valor de pico raramente encontrado.

a. [5] <6.4> Estime como o TCO por servidor mudará se a potência de servidor anunciada for de 200 watts e o custo de US$ 3.000.

b. [5] <6.4> Considere também uma opção com maior consumo, porém mais barato, cuja potência seja de 300 watts e custe US$ 2.000.

c. [10] <6.4> Como o TCO por servidor mudará se o uso médio de energia real dos servidores for de somente 70% da potência anunciada?

6.8 [15/10] <6.2, 6.4> Uma suposição, no modelo TCO, é de que a carga crítica da instalação é fixa, e a quantidade de servidores atende a essa carga crítica. Na verdade, devido às variações de consumo de energia de servidor com base na carga, a potência crítica usada por uma instalação pode variar a qualquer momento. Inicialmente, os operadores devem provisionar o datacenter com base em seus recursos energéticos críticos e numa estimativa de quanta energia é usada pelos componentes do datacenter.

a. [15] <6.2, 6.4> Estenda o modelo de TCO para inicialmente provisionar um WSC baseado em um servidor com uma potência anunciada de 300 watts, mas também calcule a energia crítica mensal real usada e o TCO, supondo que o servidor tem, em média, 40% de utilização e 225 watts. Quanta capacidade não é utilizada?

b. [10] <6.2, 6.4> Repita este exercício com um servidor de 500 watts que tenha, em média, 20% de utilização e 300 watts.

6.9 [10] <6.4, 6.5> Muitas vezes, WSCs são usados de modo interativo com os usuários finais, como mencionado na Seção 6.5. Esse uso interativo às vezes leva a flutuações durante o dia, com picos correlacionados a períodos de tempo específicos. Por exemplo, para locações da Netflix, há um pico durante os períodos noturnos entre 20-22 h. A totalidade desses efeitos de hora do dia é significativa. Compare o TCO por servidor de um datacenter com capacidade de atender a utilização às 4 h em comparação às 21 h.

6.10 [Discussão/15] <6.4, 6.5> Discuta algumas opções para utilizar melhor os servidores em excesso fora do horário de pico ou opções para poupar custos. Dada a natureza interativa dos WSCs, quais são alguns dos desafios enfrentados para reduzir agressivamente o uso de energia?

6.11 [Discussão/25] <6.4,6.6> Proponha um modo possível de melhorar o TCO concentrando-se em reduzir a potência do servidor. Quais são os desafios de avaliar sua proposta? Estime as melhoras de TCO com base nessa proposta. Quais são as vantagens e desvantagens?

Exercícios

6.12 [10/10/10] <6.1> Um dos importantes facilitadores dos WSCs é o amplo paralelismo em nível de requisição, em contraste ao paralelismo em nível de instrução ou thread. Esta questão explora a implicação de diferentes tipos de paralelismo na arquitetura de computadores e no projeto de sistemas.

- **a.** [10] <6.1> Discuta cenários em que melhorar o paralelismo em nível de instrução ou thread proporcionaria maiores benefícios do que os alcançáveis através do paralelismo em nível de requisição.
- **b.** [10] <6.1> Quais são as implicações, em projeto de software, de aumentar o paralelismo em nível de requisição?
- **c.** [10] <6.1> Quais são as potenciais desvantagens de aumentar o paralelismo em nível de requisição?

6.13 [Discussão/15/15] <6.2> Quando um provedor de serviços de computação em nuvem recebe serviços consistindo em múltiplas máquinas virtuais (VMs) (p. ex., um serviço MapReduce), existem muitas opções de escalonamento. As VMs podem ser escalonadas de modo round-robin para serem espalhadas por todos os processadores e servidores disponíveis ou podem ser consolidadas para usar o mínimo de processadores possíveis. Usando essas opções de escalonamento, se um serviço com 24 VMs tivesse sido submetido e 30 processadores estivessem disponíveis na nuvem (cada um capaz de rodar até 3 VMs), o round-robin usaria 24 processadores, enquanto o escalonamento consolidado usaria oito processadores. O escalonador também poderia encontrar núcleos de processador disponíveis em diferentes escopos: soquete, servidor, rack e um array de racks.

- **a.** [Discussão] <6.2> Supondo que os serviços submetidos sejam todos cargas de trabalho pesadas computacionalmente, possivelmente com diferentes requisitos de largura de banda de memória, quais são os prós e contras do round-robin em comparação ao escalonamento consolidado em termos de custos de energia e refrigeração, desempenho e confiabilidade?
- **b.** [15] <6.2> Supondo que os serviços submetidos tenham todos cargas de trabalho pesadas de E/S, quais são os prós e contras do round-robin em comparação ao escalonamento consolidado em diferentes escopos?
- **c.** [15] <6.2> Supondo que os serviços submetidos tenham todos cargas de trabalho pesadas de rede, quais são os prós e contras do round-robin em comparação ao escalonamento consolidado em diferentes escopos?

6.14 [15/15/10/10] <6.2, 6.3> O MapReduce permite que grandes quantidades de paralelismo tendo tarefas independentes de dados sejam executadas em múltiplos nós, muitas vezes usando hardware comercial comum. Entretanto, há limites para o nível de paralelismo. Por exemplo, por redundância, o MapReduce vai gravar blocos de dados em múltiplos nós, consumindo disco e possivelmente largura de rede. Suponha um tamanho total de conjunto de dados de 300 GB, uma largura de banda de rede de 1 Gb/s, uma taxa de map de 10 s/GB e um taxa de reduce de 20 s/GB. Suponha também que 30% dos dados devem ser lidos a partir dos nós remotos e que cada arquivo de saída é gravado em dois outros nós para redundância. Use a Figura 6.6 para todos os outros parâmetros.

a. [15] <6.2, 6.3> Suponha que todos os nós estejam no mesmo rack. Qual é o tempo de execução esperado com cinco nós? 10 nós? 100 nós? 1.000 nós? Discuta os gargalos em cada tamanho de nó.

b. [15] <6.2, 6.3> Suponha que existam 40 nós por rack e que qualquer leitura/escrita remota tenha uma chance igual de ir para qualquer nó. Qual é o tempo de execução esperado com 100 nós? E com 1.000 nós?

c. [10] <6.2, 6.3> Uma consideração importante é minimizar o movimento dos dados o máximo possível. Dada a significativa redução de velocidade da mudança de local para rack para acessos de arrays, o software deverá ser fortemente otimizado para maximizar a localização. Suponha que existam 40 nós por rack e 1.000 nós sejam usados no serviço MapReduce. Qual será o tempo de execução se os acessos remotos estiverem dentro do mesmo rack em 20% do tempo? Em 50% do tempo? Em 80% do tempo?

d. [10] <6.2, 6.3> Dado o simples programa MapReduce na Seção 6.2, discuta algumas otimizações possíveis para maximizar a localidade da carga de trabalho.

6.15 [20/20/10/20/20/20] <6.2> Muitas vezes, os programadores de WSC usam a replicação de dados para superar as falhas no software. O HDFs Hadoop, por exemplo, emprega replicação em três vias (uma cópia local, uma cópia remota no rack e uma cópia remota em um rack separado), mas vale a pena examinar quando tais replicações são necessárias.

a. [20] <6.2> Uma pesquisa envolvendo os participantes da Hadoop World 2010 mostrou que mais da metade dos clusters tinha 10 nós ou menos, com tamanhos de conjunto de dados de 10 TB ou menos. Usando os dados de frequência de falhas na Figura 6.1, que tipo de disponibilidade um cluster Hadoop de 10 nós tem com replicações de uma, duas ou três vias?

b. [20] <6.2> Usando os dados de frequência de falhas na Figura 6.1, que tipo de disponibilidade um cluster Hadoop de 1.000 nós tem com replicações de uma, duas ou três vias?

c. [10] <6.2> O overhead relativo da replicação varia com a quantidade de dados escritos por hora local de computação. Calcule a quantidade de tráfego de E/S e tráfego de rede adicional (em um e diversos racks) para um serviço Hadoop com 1.000 nós que organiza 1 PB de dados, em que os resultados imediatos para o embaralhamento de dados são escritos no HDFS.

d. [20] <6.2> Usando a Figura 6.6, calcule o overhead de tempo para replicações de duas e três vias. Usando as taxas de falha mostradas na Figura 6.1, compare os tempos de execução esperados para nenhuma replicação *versus* replicações de duas e três vias.

e. [20] <6.2> Agora considere um sistema de base de dados aplicando replicações em logs, supondo que cada transação acessa, em média, o disco rígido uma vez e gera 1 KB dos dados de log. Calcule o overhead de tempo

para replicações de duas e três vias. E se a transação for executada na memória e levar 10 μs?

f. [20] <6.2> Agora considere um sistema de base de dados com consistência ACID que requer duas idas e voltas da rede para confirmação em duas fases. Qual é o overhead de tempo para manter a consistência, além das replicações?

6.16 [15/15/20/15] <6.1, 6.2, 6.8> Embora o paralelismo em nível de requisição permita a muitas máquinas trabalhar em um único problema em paralelo, atingindo assim maior desempenho geral, um dos desafios será evitar dividir demais o problema. Se examinarmos esse problema no contexto dos acordos de nível de serviço (SLAs), usando tamanhos menores de problemas através de maior particionamento, atingir o SLA-alvo pode requerer maior esforço. Suponha um SLA de 95% de pesquisas respondidas em 0,5 s ou mais rápido, e uma arquitetura paralela similar ao MapReduce que pode lançar múltiplos serviços redundantes para atingir o mesmo resultado. Para as questões a seguir, considere a curva pesquisa-tempo de resposta mostrada na Figura 6.24. A curva mostra a latência de resposta, com base no número de pesquisas por segundo, para um servidor baseline além de um servidor "pequeno" que usa um modelo de processador mais lento.

a. [15] <6.1, 6.2, 6.8> Quantos servidores são necessários para atingir esse SLA, supondo que o WSC receba 30.000 pesquisas por segundo, e a curva pesquisa-tempo de resposta mostrada na Figura 6.24? Quantos servidores "pequenos" são necessários para atingir esse SLA, dada essa curva de probabilidade de tempo de resposta? Examinando somente os custos de servidor, quão mais baratos os servidores "fracos" devem ser em relação aos servidores normais para atingir uma vantagem de custo para o SLA-alvo?

b. [15] <6.1, 6.2, 6.8> Muitas vezes, servidores "pequenos" são também menos confiáveis, devido a componentes mais baratos. Usando os números da Figura 6.1, suponha que o número de eventos devido a máquinas instáveis e memórias ruins aumente em 30%. Quantos servidores "pequenos" são necessários agora? Quão mais baratos esses servidores devem ser em relação aos servidores-padrão?

c. [20] <6.1, 6.2, 6.8> Agora suponha um ambiente de processamento de lote. Os servidores "pequenos" fornecem 30% do desempenho geral dos servidores regulares. Ainda assumindo o número de confiabilidade do Exercício 6.15, item *b*, quantos nós "fracos" são necessários para fornecer o mesmo throughput esperado de um array de 2.400 nós de servidores-padrão, supondo escalonamento perfeitamente linear de desempenho para o tamanho

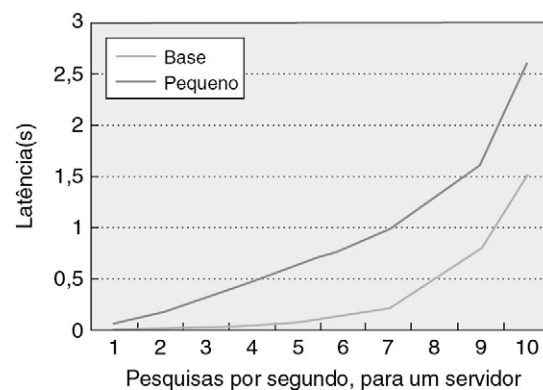



FIGURA 6.24 Curva pesquisa-tempo de resposta.

dos nós e um comprimento médio de tarefa de 10 minutos por nó? E se o escalonamento for de 85%? E de 60%?

d. [15] <<6.1, 6.2, 6.8> Muitas vezes, o escalonamento não é uma função linear, mas uma função logarítmica. Uma resposta natural poderia, em vez disso, comprar nós maiores que tenham mais poder computacional por nó para minimizar o tamanho do array. Discuta alguns dos trade-offs com essa arquitetura.

6.17 [10/10/15] <6.3, 6.8> Uma tendência em servidores de alto nível é a inclusão de memória flash não volátil na hierarquia de memória, seja através de discos de estado sólido (Solid-State Disks — SSDs), seja por meio de cartões PCI Express conectados. OS SSDs típicos têm uma largura de banda de 250 MB/s e latência de 75 μs, enquanto os cartões PCIe têm uma largura de banda de 600 MB/s e latência de 35 μs.

a. [10] Tome a Figura 6.7 e inclua esses pontos na hierarquia local de servidor. Supondo que fatores de escalonamento de desempenho idênticos, como a DRAM, sejam acessados em diferentes níveis de hierarquia, como esses dispositivos de memória flash se comparam quando acessados através do rack? Através do array?

b. [10] Discuta algumas otimizações baseadas em software que podem utilizar o novo nível da hierarquia de memória.

c. [25] Repita o item *a* supondo que cada nó tenha um cartão PCIe de 32 GB capaz de manter em cache 50% de todos os acessos de disco.

d. [15] Como discutido em "Falácias e Armadilhas" (Seção 6.8), substituir todos os discos com SSDs não é necessariamente uma estratégia efetiva em termos de custo. Considere um operador de WSC que use isso para fornecer serviços de nuvem. Discuta alguns cenários em que usar SSDs ou outra memória flash faria sentido.

6.18 [20/20/Discussão] <6.3> *Hierarquia de memória:* O armazenamento em cache é muito usado em alguns projetos de WSC para reduzir a latência, e há múltiplas opções de cache para satisfazer padrões de acesso e requerimentos variantes.

a. [20] Vamos considerar as opções de projeto para streaming de mídia da Web (p. ex., Netflix). Primeiro, precisamos estimar o número de filmes, o número de formatos de codificação por filme e usuários vendo o filme ao mesmo tempo. Em 2010, a Netflix tinha 12.000 títulos para streaming on-line, cada qual com pelo menos quatro formatos de codificação (500, 1.000, 1.600 e 2.200 kbps). Vamos supor que existam 100.000 espectadores ao mesmo tempo para todo o site e que um filme médio tenha uma hora de duração. Estime a capacidade total de armazenamento, larguras de banda de E/S e rede, e requerimentos computacionais relativos ao streaming de vídeo.

b. [20] Quais são os padrões de acesso e características de localidade de referência por usuário, por filme e através de todos os filmes? (*Dica*: Aleatório *versus* sequencial, localidade temporal e espacial boa *versus* ruim, tamanho de conjunto funcional relativamente pequeno *versus* grande.)

c. [Discussão] Que opções de armazenamento de filme existem usando DRAM, SSD e discos rígidos? Compare-os em termos de desempenho e TCO.

6.19 [10/20/20/Discussão/Discussão] <6.3> Considere um site de rede social com 100 milhões de usuários ativos postando atualizações sobre si mesmos (em texto e figuras), além de navegar e interagir com atualizações em suas redes sociais. Para proporcionar baixa latência, o Facebook e outros sites usam memcached como camada de cache antes dos itens finais de armazenamento/base de dados.

a. [10] Estime as taxas de geração de dados e de requisição por usuário e por todo o site.

b. [20] Para o site de rede social discutido aqui, quanta DRAM é necessária para hospedar seu conjunto funcional? Usando servidores com 96 GB de DRAM cada um, estime quantos acessos de memória local *versus* memória remota são necessários para gerar a home page de um usuário.

c. [20] Agora considere dois projetos de servidor memcached, um usando processadores Xeon convencionais e o outro usando núcleos menores, como processadores Atom. Como memcached requer grande memória física, mas tem pouca utilização de CPU, quais são os prós e os contras desses dois projetos?

d. [Discussão] Muitas vezes, o atual acoplamento rígido entre módulos de memória e processadores requer um aumento no número de soquetes de CPU para dar grande suporte à memória. Liste outros projetos para fornecer bastante memória física sem aumentar proporcionalmente o número de soquetes em um servidor. Compare-os com base no desempenho, consumo de energia, custos e confiabilidade.

e. [Discussão] As informações do mesmo usuário podem ser armazenadas nos servidores memcached e de armazenamento, e tais servidores podem ser hospedados fisicamente de modos diferentes. Discuta os prós e os contras do seguinte leiaute de servidor no WSC: (1) memcached colocado no mesmo servidor de armazenamento, (2) memcached e servidor de armazenamento em nós separados no mesmo rack ou (3) servidores memcached nos mesmos racks e servidores de armazenamento colocados em racks separados.

6.20 [5/5/10/10/Discussão/Discussão) <6.3, 6.6> *Rede de datacenter*: MapReduce e WSC são uma combinação poderosa para atacar o processamento de dados em grande escala. Por exemplo, em 2008 o Google organizou um petabyte (1 PB) de registros em pouco mais de seis horas usando 4.000 servidores e 48.000 discos rígidos.

a. [5] Derive a largura de banda da Figura 6.1 e texto associado. Quantos segundos leva para ler os dados na memória principal e escrever os resultados organizados de volta?

b. [5] Supondo que cada servidor tenha dois cartões de interface de rede ethernet de 1 Gb/s (NICs) e a infraestrutura de switch de WSC seja superdimensionada por um fator de 4, quantos segundos leva para organizar todo o conjunto de dados em 4.000 servidores?

c. [10] Supondo que a transferência de rede seja o gargalo de desempenho para a organização de petabytes, você pode estimar que taxa de superdimensionamento o Google tem no seu datacenter?

d. [10] Agora vamos examinar os benefícios de ter ethernet de 10 Gb/s sem superdimensionamento — por exemplo, usar uma ethernet de 10 Gb/s de 48 portas (como utilizada pelo vencedor do benchmark de organização Indy 2010, TritonSort). Quanto tempo leva para organizar o 1 PB de dados?

e. [Discussão] Compare as duas abordagens aqui: (1) a abordagem altamente fora de escala com grande taxa de oversubscription de rede; (2) um sistema de escala relativamente pequena com uma rede com grande largura de banda. Quais são seus gargalos em potencial? Quais são suas vantagens e desvantagens, em termos de escalabilidade e TCO?

f. [Discussão] Organização e muitas cargas de trabalho científicas importantes são pesadas em termos de comunicação, enquanto outras cargas de trabalho não

o são. Liste três exemplos de carga de trabalho que não se beneficiam da rede de alta velocidade. Que instâncias EC2 você recomendaria usar para essas duas classes de cargas de trabalho?

6.21 [10/25/Discussão] <6.4, 6.6> Devido à grande escala dos WSCs, é muito importante alocar corretamente os recursos de rede com base nas cargas de trabalho que se espera executar. Alocações diferentes podem ter impactos significativos sobre o desempenho e o custo total da propriedade.

a. [10] Usando os números na planilha detalhada na Figura 6.13, qual é a taxa de oversubscription em cada switch de camada de acesso? Qual será o impacto sobre o TCO se a taxa de oversubscription for cortada pela metade? E se ela for dobrada?

b. [25] Reduzir a taxa de oversubscription pode melhorar o desempenho se uma carga de trabalho for limitada pela rede. Suponha um serviço MapReduce que use 120 servidores e leia 5 TB de dados. Suponha a mesma taxa de dados de leitura/intermediários/saída da Figura 6.2 (9 set.) e use a Figura 6.6 para definir as larguras de banda da hierarquia de memória. Para a leitura de dados, suponha que 50% dos dados sejam lidos de discos remotos. Destes, 80% são lidos de dentro do rack e 20% são lidos de dentro do array. Para dados intermediários e dados de saída, suponha que 30% dos dados usem discos remotos. Destes, 90% estão dentro do rack e 10% dentro do array. Qual é a melhoria geral de desempenho quando se reduz a taxa de oversubscription pela metade? Qual será o desempenho se ela for dobrada? Calcule o TCO em cada caso.

c. [Discussão] Estamos vendo a tendência de ter mais núcleos por sistema e também a maior adoção de comunicação óptica (com possibilidade de maior largura de banda e melhor eficiência energética). Como você acha que essas e outras tendências tecnológicas emergentes vão afetar o projeto dos WSCs futuros?

6.22 [5/15/15/20/25] <6.5> *Entendendo a capacidade dos Serviços Web da Amazon*: Imagine que você seja o gerente de operações de site e infraestrutura de um site Alexa.com e está considerando usar os Serviços Web da Amazon (AWS). Que fatores você precisa considerar ao determinar se deve migrar para os AWS, que tipos de serviços e instâncias usar e quanto você economizaria nos custos? Você pode usar informações do Alexa e tráfego de site (p. ex., a Wikipédia fornece estatísticas de visualização de página) para estimar a quantidade de tráfego recebido por um grande site ou usar exemplos concretos da Web, como o seguinte exemplo da DrupalCon San Francisco 2010: http://2bits.com/sites/2bits.com/files/drupal-single-server-2.8-million-page-views-a-day.pdf. Os slides descrevem um site Alexa #3400 que recebe 2,8 milhões de page views por dia, usando um único servidor. O servidor tem dois processadores Xeon 2,5 GHz com quatro núcleos com 8 GB de DRAM e três discos rígidos SAS de 15 K RPM em uma configuração RAID1, e custa cerca de US$ 400 por mês. O site usa muito armazenamento em cache, e a utilização da CPU varia entre 50-250% (aproximadamente de 0,5-2,5 núcleos ocupados).

a. [5] Examinando as instâncias EC2 disponíveis (http://aws.amazon.com/ec2/instance-types/), que tipos de instância correspondem à atual configuração do servidor ou a excedem?

b. [15] Examinando as informações de preços do EC2 (http://aws.amazon.com/ec2/ pricing/), selecione as instâncias EC2 mais eficientes em termos de custo (são permitidas combinações) para hospedar o site no AWS. Qual é o custo mensal do EC2?

c. [15] Adicione os custos dos endereços IP e tráfego de rede à equação, e suponha que o site transfira 100 GB/dia de entrada e saída na internet. Qual é o custo mensal do site agora?

d. [20] O AWS também oferece a Microinstância gratuitamente por um ano para novos clientes e 15 GB de largura de banda para tráfego de entrada e saída através do AWS. Com base na sua estimativa de tráfego médio e de pico da web server do seu departamento, você pode hospedá-lo gratuitamente no AWS?

e. [25] Um site muito maior, Netflix.com, também migrou sua infraestrutura de streaming e codificação para o AWS. Com base nas características de serviço, que serviços AWS poderiam ser usados pela Netflix e para que objetivos?

6.23 [Discussão/Discussão/20/20/Discussão] <6.4> A Figura 6.12 mostra o impacto do tempo de resposta percebido do usuário sobre a receita, e motiva a necessidade de alcançar alto throughput ao mesmo mantendo baixa latência.

a. [Discussão] Usando a busca na Web como exemplo, quais são os modos possíveis de reduzir a latência de pesquisa?

b. [Discussão] Que estatísticas de monitoramento você pode coletar para ajudar a entender onde o tempo é gasto? Como você planeja implementar tal ferramenta de monitoramento?

c. [20] Supondo que o número de acessos de disco por busca siga uma distribuição normal, com uma **[média de 2 de desvio-padrão de 3]**, que tipo de latência de acesso de disco será necessária para satisfazer uma SLA de latência de 0,1s para 95% das pesquisas?

d. [20] A cache na memória pode reduzir as frequências de eventos de longa latência (p. ex., acessar discos rígidos). Supondo uma taxa de acerto constante de 40%, latência de acerto de 0,05 s e latência de perda de 0,2 s, o armazenamento em cache ajuda a atender uma latência de SLA de 0,1 para 95% das pesquisas?

e. [Discussão] Quando o conteúdo na cache pode se tornar antigo ou mesmo inconsistente? Com que frequência isso pode acontecer? Como você pode detectar e invalidar esse conteúdo?

6.24 (15/15/20) <6.4> A eficiência das unidades de alimentação (PSUs) típicas varia conforme a carga muda. Por exemplo, a eficiência de PSU pode ser de cerca de 80% com carga de 40% (p. ex., 40 watts de saída de uma PSU de 100 watts), 75% quando a carga está entre 20-40% e 65% quando a carga está abaixo de 20%.

a. [15] Considere um servidor com consumo de energia proporcional cujo consumo de energia real seja proporcional à utilização da CPU, com uma curva de utilização como a mostrada na Figura 6.3. Qual é a eficiência média da PSU?

b. [15] Suponha que o servidor empregue redundância 2N para as PSUs (ou seja, dobra o número de PSUs) para garantir alimentação estável quando uma PSU falha. Qual é a eficiência média da PSU?

c. [20] Os fornecedores de servidores blade usam um conjunto compartilhado de PSUs para fornecer redundância, mas também para combinar dinamicamente o número de PSUs com o consumo real de energia do servidor. O invólucro HP c7000 usa até seis PSUs para um total de 16 servidores. Nesse caso, qual é a eficiência de PSU média para o invólucro do servidor com a mesma curva de utilização?

6.25 [5/Discussão/10/15/Discussão/Discussão/Discussão] <6.4> *Encalhe de potência* é um nome usado para se referir à capacidade de potência fornecida, mas não usada em um datacenter. Considere que os dados apresentados na Figura 6.25 (Fan, Weber e Barroso, 2007) para diferentes grupos de máquinas. (Observe que esse artigo chama de "cluster" o que temos nos referidos como "array" neste capítulo.)

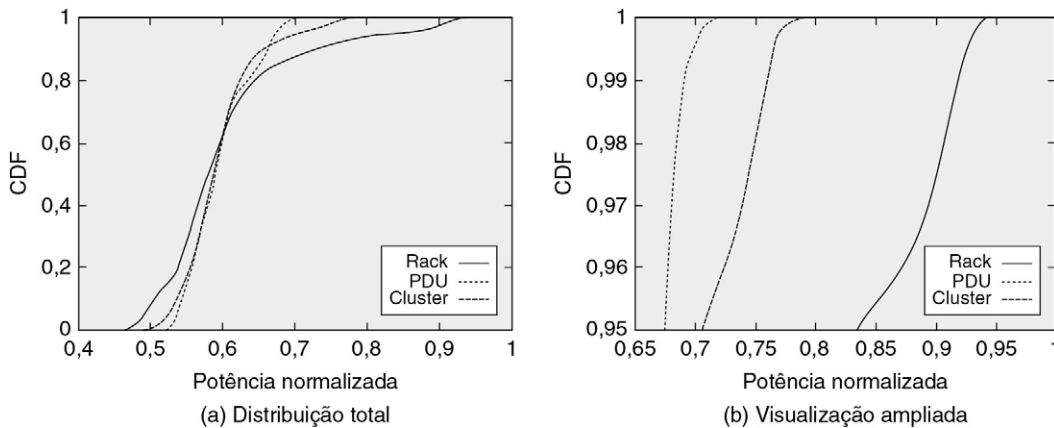



FIGURA 6.25 Função de distribuição cumulativa (Cumulative Distribution Function — CDF) de um datacenter real.

a. [5] Qual é a potência encalhada (1) no nível do rack, (2) no nível da unidade de distribuição de energia e (3) no nível do array (cluster)? Quais são as tendências com superdimensionamento de capacidade de potência em grupos maiores de máquina?

b. [Discussão] O que você acha que causa as diferenças entre encalhamento de potência em diferentes grupos de máquinas?

c. [10] Considere uma coleção de máquinas em nível de array onde o total de máquinas nunca usa mais de 72% da potência agregada (*i. e.*, às vezes, também é chamado razão entre o uso pico de soma e soma de picos). Usando o modelo de custo no estudo de caso, calcule as economias de custo ao comparar um datacenter provisionado para o pico de capacidade e um provisionado para uso real.

d. [15] Suponha que o projetista do datacenter escolha incluir servidores adicionais no nível de array para tirar vantagem da potência encalhada. Usando o exemplo de configuração e suposições no item *a*, calcule quantos servidores podem ser incluídos no computador em escala warehouse para o mesmo provisionamento total de potência.

e. [Discussão] O que é necessário para tornar a otimização do item *d* em uma implementação real? (*Dica*: Pense no que precisa acontecer para limitar o consumo de energia no caso raro em que todos os servidores no array são usados no pico de potência.)

f. [Discussão] Dois tipos de políticas podem ser imaginados para gerenciar os limites de consumo de energia (Ranganathan *et al.*, 2006): (1) políticas preventivas nas quais os orçamentos de energia sejam predeterminados ("Não suponha que você pode usar mais energia. Pergunte antes!") ou (2) políticas reativas em que os orçamentos de energia sejam estrangulados no caso de uma violação do orçamento de energia ("Use quanta energia precisar até dissermos que você não pode mais!"). Discuta os trade-offs entre essas abordagens e quando você deve usar cada tipo.

g. [Discussão] O que acontecerá à potência encalhada total se os sistemas se tornarem mais proporcionais energeticamente (suponha cargas de trabalho similares às da Figura 6.4)?

6.26 [5/20/Discussão] <6.4, 6.7> A Seção 6.7 discutiu o uso das fontes de bateria por servidor no projeto Google. Vamos examinar as consequências desse projeto.

a. [5] Suponha que o uso de uma bateria como UPS de nível de minisservidor tenha eficiência de 99,99% e elimine a necessidade de um UPS para toda a instalação que tenha somente 92% de eficiência. Considere que a troca de subestação tenha eficiência de 99,7% e que a eficiência da PDU, estágios de redução e outros breakers elétricos sejam de 98%, 98% e 99%, respectivamente. Calcule as melhorias de eficiência geral da infraestrutura de fornecimento de energia advindas do uso de uma bateria back-up por servidor.

b. [20] Suponha que o UPS seja de 10% do custo do equipamento de TI. Usando o restante das suposições do modelo de custo no estudo de caso, qual será o ponto de break-even para os custos da bateria (como uma fração do custo de um único servidor) no qual o custo total da propriedade de uma solução baseada em bateria será melhor do que o custo de um UPS para toda a instalação?

c. [Discussão] Quais são os outros trade-offs entre essas duas abordagens? Em particular, como você acha que o modelo de gerenciamento e falhas vai mudar entre esses diferentes projetos?

6.27 [5/5/Discussão] <6.4> Para este exercício, considere uma equação simplificada para a potência operacional total de um WSC, como a seguir: potência operacional total = (1 + multiplicador de ineficiência de refrigeração) * potência de equipamento de TI.

a. [5] Considere um datacenter de 8 MW com 80% de uso de energia, custos de eletricidade de US$ 0,10 por kilowatt-hora e um multiplicador de ineficiência de refrigeração de 0,8. Compare as economias de custo de (1) uma otimização que melhore a eficiência de refrigeração e (2) uma otimização que melhore a eficiência energética do equipamento de TI em 20%.

b. [5] Qual é a melhoria em porcentagem na eficiência energética do equipamento de TI necessária para corresponder às economias de custo de uma melhoria de 20% na eficiência de refrigeração?

c. [Discussão/10] Que conclusões você pode tirar sobre a importância relativa das otimizações que se concentram na eficiência energética do servidor e na eficiência energética da refrigeração?

6.28 [5/5/Discussão] <6.4> Como discutido neste capítulo, o equipamento de refrigeração nos WSCs pode consumir muita energia. Os custos de refrigeração podem ser reduzidos gerenciando proativamente a temperatura. O posicionamento de carga de trabalho ciente da temperatura é uma otimização que foi proposta para gerenciar a temperatura a fim de reduzir os custos de refrigeração. A ideia é identificar o perfil de refrigeração de determinada sala e colocar os sistemas mais quentes nos pontos mais frios, de modo que no nível do WSC os requisitos para refrigeração geral sejam reduzidos.

a. [5] O coeficiente de desempenho (Coefficient of Performance — COP) de uma unidade CRAC é definida como a razão entre o calor removido (Q) e a quantidade de trabalho necessário (W) para remover esse calor. O COP de uma unidade CRAC aumenta com a temperatura do ar que a unidade CRAC empurra para a câmara de admissão. Se o ar retornar para a unidade CRAC a 20 graus Celsius e nós removermos 10 kW de calor com um COP de 1,9, quanta energia gastaremos na unidade CRAC? Se refrigerarmos o mesmo volume de ar, mas agora retornando a 25 graus Celsius, com um COP de 3,1, quanta energia vamos gastar na unidade CRAC?

b. [5] Considere um algoritmo de distribuição de carga de trabalho capaz de combinar bem as cargas de trabalho com os pontos frios para permitir que a unidade de ar-condicionado (CRAC) da sala dos computadores funcione com

temperaturas maiores a fim de melhorar as eficiências de refrigeração, como no exercício anterior. Quais são as economias de energia nos dois casos descritos?

c. [Discussão] Dada a escala dos sistemas WSC, o gerenciamento de energia pode ser um problema complexo e multifacetado. Otimizações para melhorar a eficiência energética podem ser implementadas no hardware e no software, no nível do sistema e no nível do cluster para o equipamento de TI ou no equipamento de refrigeração etc. É importante considerar essas interações quando se projeta uma solução de eficiência energética geral para o WSC. Considere um algoritmo de consolidação que examine a utilização e consolide diferentes classes de carga de trabalho no mesmo servidor para melhorar a utilização do servidor (isso pode fazer o servidor operar com maior eficiência energética se o sistema não for proporcional em termos de energia). Como essa otimização interagiria com um algoritmo concorrente que tentasse usar diferentes estados de energia (veja alguns exemplos em ACPI, Advanced Configuration Power Interface)? Em que outros exemplos você poderia pensar onde múltiplas otimizações poderiam entrar em conflito umas com as outras em um WSC? Como você solucionaria esse problema?

6.29 [5/10/15/20] <6.2> A proporcionalidade energética (às vezes, chamada redução de escala de energia) é o atributo do sistema de não consumir nenhuma energia quando ocioso, mas — o que é mais importante — consumir gradualmente mais energia em proporção ao nível de atividade e ao trabalho realizado. Neste exercício, vamos examinar a sensibilidade do consumo de energia para diferentes modelos de proporcionalidade energética. Os exercícios a seguir, a menos que seja mencionado o contrário, usam os dados na Figura 6.4 como padrão.

a. [5] Um modo simples de pensar na proporcionalidade energética é supor a linearidade entre a atividade e o uso de energia. Usando somente os dados de pico de potência e potência ociosa na Figura 6.4 em uma interpolação linear, plote as tendências energia-eficiência em atividades variantes. (A eficiência energética é expressa como desempenho por watt.) O que acontecerá se a potência ociosa (com 0% de atividade) for a metade da suposta na Figura 6.4? O que acontecerá se a potência ociosa for zero?

b. [10] Plote as tendências energia-eficiência através de atividades variantes, mas use os dados da coluna 3 da Figura 6.4 para variação energética. Plote a eficiência energética supondo que a potência ociosa (sozinha) seja metade da suposta na Figura 6.4. Compare esses gráficos com o modelo linear do exercício anterior. Que conclusões você pode tirar sobre as consequências de se concentrar somente na potência ociosa?

c. [15] Suponha o *mix* de utilização do sistema na coluna 7 da Figura 6.4. Para simplificar, suponha uma distribuição discreta através de 1.000 servidores, com 109 servidores com 0% de utilização, 80 servidores com 10% de utilização etc. Compare o desempenho total e a energia total para esse *mix* de carga de trabalho usando as suposições nos itens *a* e *b*.

d. [20] É possível projetar um sistema com uma relação sublinear de potência *versus* carga na região dos níveis de carga entre 0-50%. Isso teria uma curva energia-eficiência com pico nas utilizações menores (à custa das maiores utilizações). Crie uma nova versão da coluna 3 da Figura 6.4 que mostre a curva energia-eficiência. Suponha o *mix* de utilização do sistema na coluna 7 da Figura 6.4. Para simplificar, suponha uma distribuição discreta através de 1.000 servidores, com 109 servidores com 0% de utilização, 80 servidores com 10% de utilização etc. Calcule o desempenho total e a energia total para esse *mix* de carga de trabalho.

Atividade (%)	0	10	20	30	40	50	60	70	80	90	100
Potência, caso A (W)	181	308	351	382	416	451	490	533	576	617	662
Potência, caso B (W)	250	275	325	340	395	405	415	425	440	445	450

FIGURA 6.26 Distribuição de potência para dois servidores.

Atividade (%)	0	10	20	30	40	50	60	70	80	90	100
N.º de servidores, casos A e B	109	80	153	246	191	115	51	21	15	12	8
N.º de servidores, caso C	504	6	8	11	26	57	95	123	76	40	54

FIGURA 6.27 Distribuições de utilização através do cluster, sem e com consolidação.

6.30 [15/20/20] <6.2, 6.6> Este exercício ilustra as interações dos modelos de proporcionalidade de energia com otimizações como projetos de consolidação de servidor e servidor eficiente em termos energéticos. Considere os cenários mostrados na Figura 6.26 e na Figura 6.27.

a. [15] Considere dois servidores com as distribuições de potência mostradas na Figura 6.26: caso A (o servidor considerado na Figura 6.4) e caso B (um servidor menos proporcional energeticamente, porém mais eficiente em termos de energia do que o caso A). Suponha o *mix* de utilização do sistema da coluna 7 da Figura 6.4. Para simplificar, suponha uma distribuição discreta através de 1.000 servidores, com 109 servidores com 0% de utilização, 80 servidores com 10% de utilização etc., como mostrado na linha 1 da Figura 6.27. Suponha uma variação de desempenho baseada na coluna 2 da Figura 6.4. Compare o desempenho total e a energia total para esse *mix* de carga de trabalho para os dois tipos de servidor.

b. [20] Considere um cluster de 1.000 servidores com dados similares aos dados mostrados na Figura 6.4 (e resumidos nas primeiras linhas das Figuras 6.26 e 6.27). Quais são o desempenho e a energia totais para o *mix* de carga de trabalho com essas suposições? Agora suponha que sejamos capazes de consolidar as cargas de trabalho para modelar a distribuição mostrada no caso C (segunda linha da Figura 6.27). Quais são o desempenho e a energia totais agora? Como a energia total se compara com um sistema que tenha um modelo linear de proporcionalidade energética com potência ociosa de zero e pico de potência de 662 watts?

c. [20] Repita o item *b* com o modelo de energia do servidor B e compare com os resultados do item *a*.

6.31 [10/Discussão] <6.2, 6.4, 6.6> *Tendências de proporcionalidade energética em nível de sistema*: Considere os seguintes detalhamentos do consumo de energia de um servidor:

CPU, 50%; memória, 23%; discos, 11%; rede/outros, 16% CPU, 33%; memória, 30%; discos, 10%; rede/outros, 27%

a. [10] Considere uma faixa de potência dinâmica de 3,0x para a CPU (ou seja, em que o consumo de energia da CPU ociosa seja um terço do seu consumo de energia no pico). Suponha que a faixa dinâmica dos sistemas de memória, discos e rede/outros acima sejam respectivamente 2,0x, 1,3x e 1,2x. Qual é a faixa dinâmica geral para o sistema total nesses dois casos?

Camada 1	Caminho simples para distribuições de energia e refrigeração, sem componentes redundantes	99%
Camada 2	Redundância (N + 1) = dois caminhos de distribuição de energia e refrigeração	99,7%
Camada 3	Redundância (N + 2) = três caminhos de distribuição de energia e refrigeração para o uptime, mesmo durante a manutenção	99,98%
Camada 4	Dois caminhos ativos de distribuição de energia e refrigeração, com componentes redundantes em cada caminho, para tolerar qualquer falha única de equipamento sem impactar a carga	99,995%

FIGURA 6.28 Visão geral das classificações das camadas do datacenter. *(Adaptado de Pitt Turner IV* et al., *2008.)*

b. [Discussão/10] O que você pode aprender com os resultados do item *a*? Como poderíamos atingir melhor proporcionalidade energética no nível de sistema? (*Dica*: A proporcionalidade energética em nível de sistema não pode ser atingida somente através de otimizações de CPU; em vez disso, requer melhorias em todos os componentes.)

6.32 [30] <6.4> Pitt Turner IV *et al.* (2008) apresentaram uma boa visão geral das classificações das camadas de datacenter. As classificações de camada definem o desempenho da infraestrutura do site. Para simplificar, considere as principais diferenças como mostradas na Figura 6.25 (adaptada de Pitt Turner IV *et al.*, 2008). Usando o modelo TCO no estudo de caso, compare as implicações de custo das diferentes camadas mostradas.

6.33 [Discussão] <6.4> Com base nas observações na Figura 6.13, o que você poderia dizer qualitativamente sobre os trade-offs entre as perdas de receita de tempo inativo e os custos para o tempo ativo?

6.34 [15/Discussão] <6.4> Alguns estudos recentes definiram uma métrica chamada TPUE, que significa "PUE verdadeiro" (true PUE) ou "PUE total" (total PUE). O TPUE é definido como PUE*SPUE. O PUE, eficiência da utilização de energia, é definida na Seção 6.4 como a razão entre o consumo de energia total da instalação e o consumo total de energia do equipamento de TI. O SPUE, ou PUE de servidor, é uma nova métrica análoga ao PUE, mas aplicada para o equipamento computacional, e é definido como a razão entre a potência total de entrada do servidor e sua potência útil, em que a potência útil é definida como a energia consumida pelos componentes eletrônicos envolvidos diretamente no cálculo: placa-mãe, discos, CPUs, DRAM, cartões de E/S, e assim por diante. Em outras palavras, a métrica SPUE captura ineficiências associadas com as fontes de energia, reguladores de voltagem e ventoinhas em um servidor.

a. [15] <6.4> Considere um projeto que use maior temperatura de alimentação para as unidades CRAC. A eficiência da unidade CRAC é aproximadamente uma função quadrática da temperatura, portanto, esse projeto melhora o PUE geral em cerca de 7% (considere um PU base de 1,7x). Entretanto, a maior temperatura no nível de servidor ativa o controlador da ventoinha na placa para operar a ventoinha a velocidades muito maiores. A potência da ventoinha é uma função quadrada da velocidade, e a maior velocidade da ventoinha leva a uma degradação do SPUE. Considere um modelo de potência de ventoinha:

$$\text{Potência da ventoinha} = 284 * ns * ns * ns - 75 * ns * ns,$$

onde n*s* é a velocidade normalizada da ventoinha = velocidade da ventoinha em RPM/18.000

e uma potência base de servidor de 350 W. Calcule o SPU se a velocidade da ventoinha aumentar de (1) 10.000 rpm para 12.500 rpm e (2) de 10.000 rpm para 18.000 rpm. Compare o PUE e o TPUE nos dois casos. (Para simplificar, ignore as ineficiências com entregas de energia no modelo SPUE.)

b. [Discussão] O item *a* ilustra que, embora o PUE seja uma excelente métrica para capturar o overhead da instalação, não captura as ineficiências dentro do próprio equipamento TI. Você pode identificar outro projeto em que o TPUE seja potencialmente menor do que o PUE? (*Dica*: Ver o Exercício 6.26.)

6.35 [Discussão/30/Discussão] <6.2> Dois benchmarks recém-lançados fornecem um bom ponto de partida para o controle da eficiência energética nos servidores — o benchmark SPCPower ssj2008 (disponível em www.spec.org/power_ssj2008/) e a métrica JouleSort (disponível em http://sortbenchmark.org/).

a. [Discussão] <6.2> Procure as descrições dos dois benchmarks. Em que eles são similares? Em que são diferentes? O que você faria para melhorar esses benchmarks para alcançar o objetivo de melhorar a eficiência energética de um WSC?

b. [30] <6.2> O JouleSort mede a energia total do sistema para realizar uma organização fora do núcleo e tenta derivar uma medida que permita a comparação de sistemas variando entre dispositivos embarcados e supercomputadores. Procure a descrição da medida JouleSort em http://sortbenchmark.org. Faça o download (procure uma versão disponível para o público) do algoritmo de organização em diferentes classes de máquinas — um laptop, um PC, um telefone celular etc. — ou com diferentes configurações. O que você pode aprender com as classificações JouleSort para diferentes configurações?

c. [Discussão] <6.2> Considere o sistema com a melhor classificação JouleSort a partir dos experimentos anteriores. Como você poderia melhorar a eficiência energética? Por exemplo, tente reescrever o código de organização para melhorar a classificação JouleSort.

6.36 [10/10/15] <6.1, 6.2> A Figura 6.1 é uma listagem de indisponibilidades em um array de servidores. Quando lidamos com a grande escala dos WSCs, é importante equilibrar o projeto do cluster e as arquiteturas de software para atingir o tempo ativo desejado sem incorrer em custos significativos. Esta questão explora as implicações de atingir a disponibilidade somente através de hardware.

a. [10] <6.1, 6.2> Supondo que um operador queira atingir 95% de disponibilidade somente através de melhorias no hardware de servidor, quantos eventos de cada tipo precisariam ser reduzidos? Por enquanto, suponha que as "quedas" de servidores individuais sejam completamente tratadas através de máquinas redundantes.

b. [10] <6.1, 6.2> Como a resposta ao item *a* mudaria se as quedas dos servidores individuais fossem tratadas por redundância em 50% das vezes? E em 20% do tempo? E nunca?

c. [15] <6.1, 6.2> Discuta a importância da redundância de software para atingir um alto nível de disponibilidade. Se um operador de WSC considerasse comprar máquinas mais baratas, porém 10% menos confiáveis, que implicações isso teria na arquitetura de software? Quais são os desafios associados com a redundância de software?

6.37 [15] <6.1, 6.8> Procure os preços atuais da DRAM DDR3 padrão em comparação à DRAM DDR3 que tenha código de correção de erro (ECC). Qual é o aumento no preço por bit para atingir a maior disponibilidade proporcionada

pelo ECC? Usando somente os preços da DRAM e os dados fornecidos na Seção 6.8, qual é o tempo ativo por dólar de um WSC com DRAM sem ECC *versus* uma DRAM com ECC?

6.38 [5/Discussão] <6.1> Problemas de confiabilidade e gerenciamento de WSC:

a. [5] Considere um cluster de servidores custando US$ 2.000 cada um. Supondo uma taxa anual de falha de 5%, uma média de uma hora de tempo de serviço por reparo e as partes de substituição requerendo 10% do custo do sistema por falha, qual é o custo anual de manutenção por servidor? Suponha uma taxa horária de US$ 100 por hora para um técnico de serviço.

b. [Discussão] Comente as diferenças entre esse modelo de gerenciamento *versus* o de um datacenter empresarial tradicional com um grande número de aplicações de tamanho médio, cada uma rodando na sua própria infraestrutura dedicada de hardware.

Princípios e exemplos de conjuntos de instruções

An Some o número no local de armazenamento *n* no acumulador.
En Se o número no acumulador for maior ou igual a zero, execute a próxima ordem que está no local de armazenamento *n*; caso contrário, proceda serialmente.
Z Pare a máquina e toque a campainha de aviso.

Wilkes e Renwick
Seleção da Lista das 18 Instruções de Máquina para o EDSAC (1949)

A.1 INTRODUÇÃO

Neste apêndice, nos concentraremos na arquitetura do conjunto de instruções — a parte do computador visível ao programador ou projetista de compiladores. A maior parte deste material deverá servir de revisão para os leitores; nós o incluímos aqui apenas como base. Este apêndice apresenta a grande variedade de alternativas de projeto disponíveis para o projetista do conjunto de instruções. Quatro tópicos serão enfocados: 1) vamos apresentar uma taxonomia alternativa de conjunto de instruções fixas e uma avaliação qualitativa das vantagens e desvantagens dos vários métodos; 2) vamos apresentar e analisar algumas medidas de um conjunto de instruções que são bastante independentes de um conjunto de instruções específico; 3) vamos tratar do problema das linguagens e dos compiladores e de sua relação com a arquitetura do conjunto de instruções; 4) por fim, a seção "Juntando tudo" vai mostrar como essas ideias são refletidas no conjunto de instruções MIPS, típico das arquiteturas RISC. Concluiremos com as falácias e as armadilhas do projeto do conjunto de instruções.

Para ilustrar melhor os princípios, o Apêndice K mostra quatro exemplos de arquiteturas de RISC de finalidade geral (MIPS, PowerPC, Precision Architecture, SPARC), quatro processadores de RISC embutidos (ARM, Hitachi SH, MIPS 16, Thumb) e três arquiteturas mais

antigas (80x86, IBM 360/370 e VAX). Antes de discutirmos como classificar as arquiteturas, precisamos dizer algo sobre a medida do conjunto de instruções.

Ao longo deste apêndice, examinaremos uma grande variedade de medidas arquitetônicas. Obviamente, essas medidas dependem dos programas medidos e dos compiladores que estiverem fazendo as medições. Os resultados não devem ser interpretados como absolutos, pois você poderia ver dados diferentes se fizesse a medição com um compilador ou um conjunto de programas diferente. Acreditamos que as medidas apresentadas neste apêndice são razoavelmente indicativas de uma classe de aplicações típicas. Muitas dessas medidas são apresentadas usando um pequeno conjunto de benchmarks, de forma que os dados podem ser exibidos de modo lógico e as diferenças entre os programas podem ser vistas. Diante de um computador novo, um projetista provavelmente gostaria de analisar um conjunto maior de programas antes de tomar decisões sobre a arquitetura. As medidas mostradas normalmente são *dinâmicas* — ou seja, a frequência de um evento medido é ponderada de acordo com o número de vezes que o evento acontece durante a execução do programa medido.

Antes de começar a tratar dos princípios gerais, vamos revisar as três áreas de aplicação do Capítulo 1. A *computação de desktop* enfatiza o desempenho de programas com tipos de dados de inteiro e de ponto flutuante, com pouca consideração sobre o tamanho do programa ou o consumo de energia do processador. Por exemplo, o tamanho de código nunca foi informado nas cinco gerações de benchmarks SPEC. Hoje, os *servidores* são usados principalmente como banco de dados, servidor de arquivo e aplicações de Web, assim como para algumas aplicações de compartilhamento de tempo para muitos usuários. Consequentemente, o ponto flutuante é muito menos importante para o desempenho do que os inteiros e strings de caracteres, embora quase todo processador ainda inclua instruções de ponto flutuante. As *aplicações embarcadas* avaliam custo e energia; assim, o tamanho de código é importante porque menos memória é sinônimo de memória mais barata e que consome menos energia; além disso, algumas classes de instrução (como ponto flutuante) podem ser opcionais para reduzir custos de chip.

Portanto, os conjuntos de instruções para as três aplicações são muito semelhantes. Na verdade, a arquitetura MIPS que enseja este apêndice foi usada com sucesso em aplicações de desktop, servidores e embarcadas.

Uma arquitetura bem-sucedida muito diferente da RISC é a 80x86 (Apêndice K). Surpreendentemente, seu sucesso não desmente necessariamente as vantagens de um conjunto de instruções RISC. A importância comercial da compatibilidade binária com o software de PC, combinada com a abundância de transistores fornecida pela lei de Moore, levou a Intel a usar um conjunto de instruções RISC internamente enquanto aceitava um conjunto de instruções 80x86 externamente. Os microprocessadores 80x86 recentes, como o Pentium 4, usam hardware para traduzir instruções 80x86 para instruções tipo RISC e então executar as operações traduzidas dentro do chip. Eles mantêm a ilusão da arquitetura 80x86 para o programador enquanto permitem ao projetista de computador implementar um processador do tipo RISC para desempenho.

Agora que a base está preparada, começamos explorando o modo como as arquiteturas de conjunto de instruções podem ser classificadas.

A.2 CLASSIFICANDO AS ARQUITETURAS DE CONJUNTO DE INSTRUÇÕES

O tipo de armazenamento interno em um processador é a diferenciação mais básica; portanto, nesta seção, enfocaremos as alternativas para essa parte da arquitetura. As principais escolhas são: uma pilha, um acumulador ou um conjunto de registradores. Os operandos

podem ser nomeados explícita ou implicitamente: os operandos em uma *arquitetura de pilha* estão implicitamente no topo da pilha e, em uma *arquitetura de acumulador*, um operando é implicitamente o acumulador. As *arquiteturas de registradores de propósito geral* possuem apenas operandos explícitos — registradores ou locais de memória. A Figura A.1 mostra um diagrama de bloco dessas arquiteturas, e a Figura A.2 mostra como a

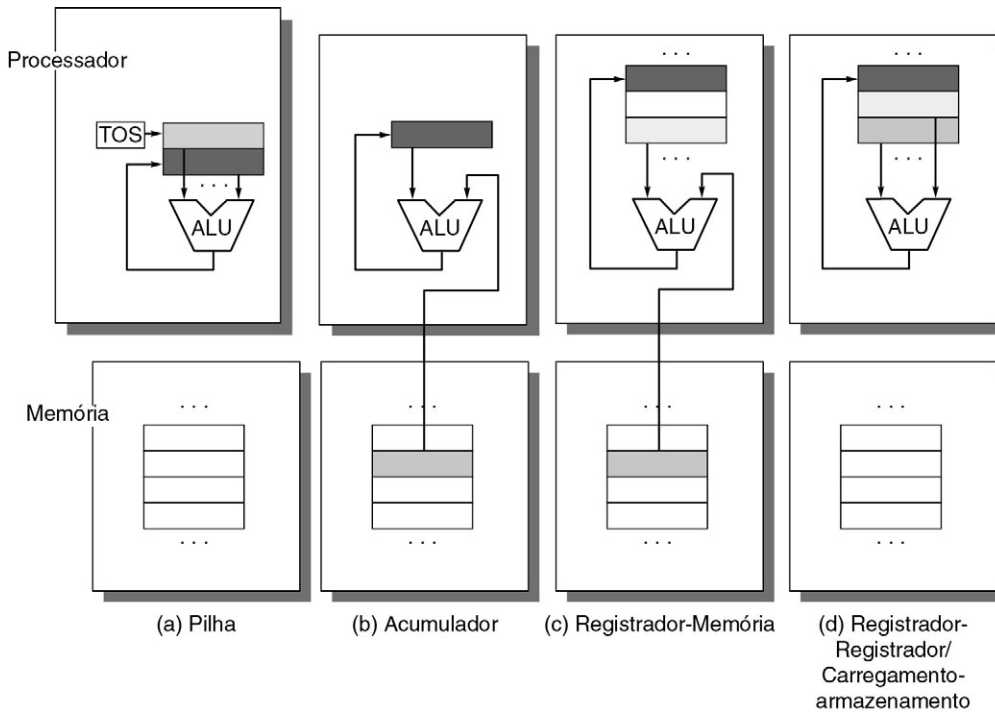



FIGURA A.1 **Locais de operandos para as quatro classes de arquitetura de conjunto de instruções.** As setas indicam se o operando é uma entrada ou o resultado da operação da ULA ou se é ao mesmo tempo uma entrada e o resultado. Os sombreados claros indicam entradas, e o sombreado mais escuro indica o resultado. Em (a), um registrador Top Of Stack (TOS) aponta para o operando de entrada superior, que é combinado com o operando abaixo. O primeiro operando é removido da pilha, o resultado assume o lugar do segundo operando, e o TOS é atualizado para apontar para o resultado. Todos os operandos estão implícitos. Em (b), o acumulador é tanto um operando de entrada implícito quanto um resultado. Em (c), um operando de entrada é um registrador, um está em memória e o resultado vai para um registrador. Todos os operandos são registradores em (d) e, como a arquitetura de pilha, só pode ser transferido para a memória através de instruções separadas: push ou pop para (a) e carregamento ou armazenamento para (d).

Pilha	Acumulador	Registrador (registrador-memória)	Registrador (carregamento-armazenamento)
`Push A`	`Load A`	`Load R1,A`	`Load R1,A`
`Push B`	`Add B`	`Add R3,R1,B`	`Load R2,B`
`Add`	`Store C`	`Store R3,C`	`Add R3,R1,R2`
`Pop C`			`Store R3,C`

FIGURA A.2 **A sequência de código para C = A + B para quatro classes de conjuntos de instruções.** Observe que a instrução Add possui operandos implícitos para arquiteturas de pilha e de acumulador, e operandos explícitos para arquiteturas de registrador. Considera-se que A, B e C pertençam à memória e que os valores de A e B não podem ser destruídos. A Figura A.1 mostra a operação Add para cada classe da arquitetura.

sequência de código C = A + B normalmente se pareceria nessas três classes de conjuntos de instruções. Os operandos explícitos podem ser acessados diretamente da memória ou precisar ser primeiramente carregados no armazenamento temporário, dependendo da classe da arquitetura e da escolha da instrução específica.

Como mostram as figuras, na realidade, há duas classes de registradores: uma classe que pode acessar a memória como parte de qualquer instrução, chamada arquitetura *registrador-memória*, e a outra que só pode acessar a memória com instruções de carregamento (load) e armazenamento (store), chamada de arquitetura carregamento-armazenamento (load-store) ou registrador-registrador. Uma terceira classe, não encontrada em computadores modernos, mantém todos os operandos na memória e é chamada arquitetura *memória-memória*. Algumas arquiteturas de conjunto de instruções possuem mais registros do que um único acumulador, mas impõem restrições sobre os usos desses registradores especiais. Às vezes tal arquitetura é chamada *acumulador estendido* ou computador *registrador de propósito específico*.

Embora a maioria dos primeiros computadores usasse arquiteturas de pilha ou de acumulador, quase todas as novas arquiteturas projetadas após 1980 usam uma arquitetura de registrador de carregamento-armazenamento. São duas as principais razões para o surgimento dos computadores com registradores de propósito geral (GPR): 1) os registradores, assim como outras formas de armazenamento internas ao processador, são mais rápidos do que a memória; 2) para um compilador, usar registradores é mais eficiente do que empregar outras formas de armazenamento interno. Por exemplo, em um computador de registrador, a expressão (A * B) – (B * C) – (A * D) pode ser avaliada efetuando-se as multiplicações em qualquer ordem, o que pode ser mais eficiente, devido ao local dos operandos ou às preocupações com o pipelining (Cap. 2). Entretanto, em um computador de pilha, o hardware precisa avaliar a expressão apenas em uma ordem, já que os operandos são ocultos na pilha e pode ser necessário carregar um operando diversas vezes.

Mais importante ainda, podem ser usados os registradores para armazenar as variáveis. Quando as variáveis são alocadas em registradores, o tráfego de memória reduz, o programa acelera (já que os registradores são mais rápidos que a memória) e a densidade de código melhora (já que um registrador pode ser nomeado com menos bits do que um local de memória).

Como explicado na Seção A.8, os projetistas de compilador prefeririam que todos os registradores fossem equivalentes e não reservados. Os computadores mais antigos comprometem esse desejo dedicando registradores a usos especiais, diminuindo efetivamente o número de registradores de propósito geral. Se o número de registradores de propósito geral for muito pequeno, tentar alocar variáveis para registradores não será proveitoso. Em vez disso, o compilador reservará todos os registradores não comprometidos para uso na avaliação de expressões.

Quantos registradores são suficientes? A resposta, é claro, depende da eficiência do compilador. A maioria dos compiladores reserva alguns registradores para avaliação de expressões, usa alguns para passagem de parâmetros e permite que o restante seja alocado para armazenar variáveis. A tecnologia de compiladores modernos e sua capacidade de usar eficientemente um número maior de registradores levaram a um aumento no número de registradores nas arquiteturas mais novas.

Duas grandes características de conjuntos de instruções dividem as arquiteturas GPR. Essas duas características se relacionam à natureza dos operandos para uma aritmética ou lógica típica (instrução da ULA). A primeira é sobre se uma instrução da ULA tem dois ou três operandos. No formato de três operandos, a instrução contém um operando de resultado e dois operandos de origem. No formato de dois operandos, um dos operandos é, ao mesmo tempo, uma origem e um resultado para a operação. A segunda distinção entre as arquiteturas GPR se refere a quantos dos operandos podem ser endereços de memória

nas instruções da ULA. O número de operandos de memória aceito por uma instrução da ULA típica pode variar de zero a três. A Figura A.3 mostra combinações desses dois atributos com exemplos de computadores. Embora haja sete combinações possíveis, três servem para classificar quase todos os computadores existentes. Como já mencionamos, essas três são carregamento-armazenamento (também chamada registrador-registrador), registrador-memória e memória-memória.

A Figura A.4 mostra as vantagens e desvantagens de cada uma dessas alternativas. Sem dúvida, essas vantagens e desvantagens não são absolutas: elas são qualitativas e seu verdadeiro impacto depende do compilador e da estratégia de implementação. Um computador GPR com operações memória-memória poderia ser facilmente ignorado pelo compilador e usado como um computador de carregamento-armazenamento. Um dos impactos arquitetônicos mais penetrantes é sobre a codificação de instrução e o número de instruções necessárias para executar uma tarefa. Examinamos o impacto dessas alternativas arquitetônicas sobre os métodos de implementação no Apêndice C e no Capítulo 3.

Número de endereços de memória	Número máximo de operandos permitidos	Tipo de arquitetura	Exemplos
0	3	Carregamento-armazenamento/ registrador-registrador	Alpha, ARM, MIPS, PowerPC, SPARC, SuperH, TM32
1	2	Registrador-memória	IBM 360/370, Intel 80x86, Motorola 68000,TI TMS320C54x
2	2	Memória-memória	VAX (também possui formatos de três operandos)
3	3	Memória-memória	VAX (também possui formatos de dois operandos)

FIGURA A.3 Combinações típicas de operandos de memória e totais de operandos por instruções típicas da ULA com exemplos de computadores. Os computadores sem referência à memória por instrução da ULA são chamados de computadores de carregamento-armazenamento ou registrador-registrador. As instruções com vários operandos de memória por instruções típicas da ULA são chamadas de registrador-memória ou memória-memória, conforme tenham um ou mais de um operando de memória.

Tipo	Vantagens	Desvantagens
Registrador-registrador (0, 3)	Codificação de instruções simples e de tamanho fixo. As instruções são executadas com um número de clocks semelhantes (Apêndice C).	Contagem de instrução mais alta do que as arquiteturas com referências à memória nas instruções. Mais instruções e densidade de instrução mais baixa produzem programas maiores.
Registrador-memória (1, 2)	Os dados podem ser acessados sem executar uma instrução de carregamento (load) primeiro. O formato de instrução costuma ser fácil de codificar e apresenta boa densidade.	Os operandos não são equivalentes, já que um operando de origem em uma operação binária é destruído. Codificar um número de registrador e um endereço de memória em cada instrução pode restringir o número de registradores. O número de clocks por instrução varia conforme o local do operando.
Memória-memória (2, 2) ou (3, 3)	Mais compacto. Não desperdiça registradores para temporários.	Grande variação no tamanho de instrução, especialmente para instruções de três operandos. Além disso, grande variação no trabalho por instrução. Os acessos à memória criam gargalos de memória. (Não usado atualmente.)

FIGURA A.4 Vantagens e desvantagens dos três tipos mais comuns de computadores registradores de finalidade geral. A notação (*m, n*) significa *m* operandos de memória e *n* operandos totais. Em geral, os computadores com menos alternativas simplificam a tarefa do compilador, já que existem menos decisões para o compilador tomar (Seção A.8). Os computadores com grande variedade de formatos de instrução flexíveis reduzem o número de bits necessários para codificar o programa. O número de registradores também afeta o tamanho da instrução, já que $\log_2$ (número de registradores) é necessário para cada especificador de registrador em uma instrução. Portanto, dobrar o número de registrador consome 3 bits extras para uma arquitetura registrador-registrador ou aproximadamente 10% de uma instrução de 32 bits.

Resumo: classificando as arquiteturas de conjunto de instruções

Aqui e no final das Seções A.3 a A.8, resumimos as características que esperaríamos encontrar em uma nova arquitetura de conjunto de instruções, construindo os fundamentos para a arquitetura MIPS introduzida na Seção A.9. A partir desta seção, devemos claramente esperar o uso de registradores de propósito geral. A Figura A.4, combinada com o Apêndice C sobre pipelining, leva à expectativa de uma versão carregamento-armazenamento (load-store) de uma arquitetura de registrador de propósito geral.

Com a classe da arquitetura discutida, o próximo tópico é o endereçamento dos operandos.

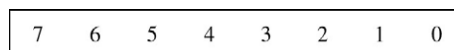
A.3 ENDEREÇAMENTO DE MEMÓRIA

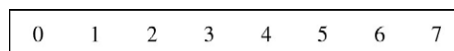
Quer a arquitetura seja carregamento-armazenamento, quer permita que qualquer operando seja uma referência à memória, ela precisa definir como os endereços de memória são interpretados e como são especificados. As medidas apresentadas aqui são, em grande parte — mas não completamente — independentes do computador. Em alguns casos, as medidas são significativamente afetadas pela tecnologia do compilador. Essas medidas foram feitas usando um compilador de otimização, já que a tecnologia de compiladores desempenha um papel vital.

Interpretando endereços de memória

Como um endereço de memória é interpretado? Ou seja, que objeto é acessado como uma função do endereço e do comprimento? Todos os conjuntos de instruções discutidos neste livro são endereçados por byte e fornecem acesso para bytes (8 bits), meias palavras (16 bits) e palavras (32 bits). A maioria dos computadores também fornece acesso para palavras duplas (64 bits).

Existem duas convenções diferentes para ordenar os bytes dentro de um objeto maior. A ordem de bytes *Little Endian* coloca o byte cujo endereço é "x.. . x000" na posição menos significativa da palavra dupla (a extremidade de menor ordem). Os bytes são numerados:

7	6	5	4	3	2	1	0

A ordem de byte *Big Endian* coloca o byte cujo endereço é "x.. . x000" na posição mais significativa da palavra dupla (extremidade de maior ordem). Os bytes são numerados:

0	1	2	3	4	5	6	7

Ao operar dentro de um computador, a ordem de byte costuma ser imperceptível — apenas programas que acessam os mesmos locais, como palavras e bytes, por exemplo, podem notar a diferença. Entretanto, a ordem de bytes é um problema ao trocar dados entre computadores com ordenações diferentes. A ordenação Little Endian também não corresponde à ordenação normal de palavras quando strings são comparadas. As strings aparecem ao "OIRÁRTNOC" (ao contrário) nos registros.

Um segundo aspecto da memória é que, em muitos computadores, os acessos a objetos maiores que um byte devem estar *alinhados*. Um acesso a um objeto de tamanho s bytes no endereço de byte A é alinhado se $A \bmod s = 0$. A Figura A.5 mostra os endereços em que um acesso está alinhado ou desalinhado.

Por que alguém projetaria um computador com restrições de alinhamento? O desalinhamento causa complicações de hardware, já que a memória normalmente é alinhada

em um múltiplo de um limite de uma palavra ou palavra dupla. Um acesso à memória desalinhado, portanto, pode envolver múltiplas referências de memória alinhadas. Assim, até mesmo em computadores que permitem acesso desalinhado, os programas com acessos alinhados são executados mais rapidamente.

Mesmo se os dados estiverem alinhados, o suporte a byte, meia palavra e palavra exige uma rede de alinhamento para alinhar bytes, meias palavras e palavras em registradores de 64 bits. Por exemplo, na Figura A.5, suponha a leitura de um byte de um endereço com seus 3 bits de baixa ordem tendo o valor 4. Precisaremos deslocar 3 bytes para a direita para alinhar o byte com o lugar correto em um registrador de 64 bits. Dependendo da instrução, o computador também pode precisar estender o sinal. Os armazenamentos são fáceis: apenas os bytes endereçados na memória podem ser alterados. Em alguns computadores, uma operação de byte, meia palavra e palavra não afeta a parte superior de um registrador. Embora todos os computadores discutidos neste livro permitam acessos de byte, meia palavra e palavra à memória, apenas o IBM 360/370, o Intel 80x86 e o VAX aceitam operações da ULA em operandos de registradores menores que a largura total.

Agora que discutimos as interpretações alternativas dos endereços de memória, podemos discutir as maneiras como os endereços são especificados por instruções, chamadas *modos de endereçamento*.

Modos de endereçamento

Dado um endereço, sabemos que bytes acessar na memória. Nesta subseção, veremos os modos de endereçamento — como as arquiteturas especificam o endereço de um objeto

<table>
<tr><th></th><th colspan="8">Valor dos 3 bits de ordem inferior do endereço de byte</th></tr>
<tr><th>Largura do objeto</th><th>0</th><th>1</th><th>2</th><th>3</th><th>4</th><th>5</th><th>6</th><th>7</th></tr>
<tr><td>1 byte (byte)</td><td>Alinhado</td><td>Alinhado</td><td>Alinhado</td><td>Alinhado</td><td>Alinhado</td><td>Alinhado</td><td>Alinhado</td><td>Alinhado</td></tr>
<tr><td>2 bytes (meia palavra)</td><td colspan="2">Alinhado</td><td colspan="2">Alinhado</td><td colspan="2">Alinhado</td><td colspan="2">Alinhado</td></tr>
<tr><td>2 bytes (meia palavra)</td><td></td><td colspan="2">Desalinhado</td><td colspan="2">Desalinhado</td><td colspan="2">Desalinhado</td><td>Desalinhado</td></tr>
<tr><td>4 bytes (palavra)</td><td colspan="4">Alinhado</td><td colspan="4">Alinhado</td></tr>
<tr><td>4 bytes (palavra)</td><td></td><td colspan="4">Desalinhado</td><td colspan="3">Desalinhado</td></tr>
<tr><td>4 bytes (palavra)</td><td colspan="2"></td><td colspan="4">Desalinhado</td><td colspan="2">Desalinhado</td></tr>
<tr><td>4 bytes (palavra)</td><td colspan="3"></td><td colspan="4">Desalinhado</td><td>Desalinhado</td></tr>
<tr><td>8 bytes (palavra dupla)</td><td colspan="8">Alinhado</td></tr>
<tr><td>8 bytes (palavra dupla)</td><td></td><td colspan="7">Desalinhado</td></tr>
<tr><td>8 bytes (palavra dupla)</td><td colspan="2"></td><td colspan="6">Desalinhado</td></tr>
<tr><td>8 bytes (palavra dupla)</td><td colspan="3"></td><td colspan="5">Desalinhado</td></tr>
<tr><td>8 bytes (palavra dupla)</td><td colspan="4"></td><td colspan="4">Desalinhado</td></tr>
<tr><td>8 bytes (palavra dupla)</td><td colspan="5"></td><td colspan="3">Desalinhado</td></tr>
<tr><td>8 bytes (palavra dupla)</td><td colspan="6"></td><td colspan="2">Desalinhado</td></tr>
<tr><td>8 bytes (palavra dupla)</td><td colspan="7"></td><td>Desalinhado</td></tr>
</table>

FIGURA A.5 Endereços alinhados e desalinhados de objetos com tamanho de byte, meia palavra, palavra e palavra dupla para computadores endereçados por byte.

Para cada exemplo desalinhado, alguns objetos exigem dois acessos à memória para serem completados. Cada objeto alinhado sempre pode ser completado em um acesso à memória, desde que a memória seja tão grande quanto o objeto. A figura mostra a memória organizada como 8 bytes de extensão. Os deslocamentos de byte que rotulam as colunas especificam os 3 bits de ordem inferior do endereço.

que elas acessarão. Os modos de endereçamento especificam constantes e registradores, além de posições na memória. Quando um local da memória é usado, o endereço de memória real especificado pelo modo de endereçamento é chamado *endereço efetivo*.

A Figura A.6 mostra todos os modos de endereçamento de dados que foram usados em computadores modernos. Imediatos ou literais normalmente são considerados modos de endereçamento de memória (ainda que o valor que acessam esteja no fluxo de instrução), embora os registradores frequentemente sejam separados, já que eles normalmente não possuem endereços de memória. Temos mantido separados os modos de endereçamento que dependem do contador de programa, chamado *endereçamento relativo ao PC*. O endereçamento relativo ao PC é usado principalmente para especificar endereços de código em instruções de transferência de controle, discutidas na Seção A.6.

A Figura A.6 mostra os nomes mais comuns para os modos de endereçamento, embora os nomes difiram entre as arquiteturas. Nessa figura e em todo o livro, usamos uma extensão para a linguagem de programação C como uma notação de descrição de hardware.

Modo de endereçamento	Instrução de exemplo	Significado	Quando é usado
Registrador	`Add R4,R3`	`Regs[R4] ← Regs[R4] + Regs[R3]`	Quando um valor está em um registrador.
Imediato	`Add R4,#3`	`Regs[R4] ← Regs[R4] + 3`	Para constantes.
Deslocamento	`Add R4,100(R1)`	`Regs[R4] ← Regs[R4] + Mem[100+Regs[R1]]`	Ao acessar variáveis locais (+ simula modos de endereçamento indireto de registrador e direto).
Indireto de registrador	`Add R4,(R1)`	`Regs[R4] ← Regs[R4] + Mem[Regs[R1]]`	Ao acessar usando um ponteiro ou um endereço calculado.
Indexado	`Add R3,(R1+R2)`	`Regs[R3] ← Regs[R3] + Mem[Regs[R1]+Regs[R2]]`	Algumas vezes útil no endereçamento de array: `R1` = base do array; `R2` = valor de índice.
Direto ou absoluto	`Add R1,(1001)`	`Regs[R1] ← Regs[R1] + Mem[1001]`	Algumas vezes útil para acessar dados estáticos; a constante do endereço pode precisar ser grande.
Indireto de memória	`Add R1,@(R3)`	`Regs[R1] ← Regs[R1] + Mem[Mem[Regs[R3]]]`	Se `R3` é o endereço de um ponteiro *p*, então o modo produz **p*.
Autoincremento	`Add R1,(R2)+`	`+ Mem[Regs[R2]]` `Regs[R2] ← Regs[R2] + d`	Útil para percorrer os arrays passo a passo dentro de um loop. `R2` aponta para o início do array; cada referência incrementa `R2` pelo tamanho de um elemento, *d*.
Autodecremento	`Add R1,-(R2)`	`Regs[R2] ¬ Regs[R2] - d` `Regs[R1] ¬ Regs[R1] + Mem[Regs[R2]]`	Mesmo uso do autoincremento. Autodecremento/autoincremento também podem agir como push/pop para implementar uma pilha.
Escalonado	`Add R1,100(R2)[R3]`	`Regs[R1] ← Regs[R1] + Mem[100+Regs[R2] + Regs[R3] * d]`	Usado para indexar arrays. Pode ser aplicado para qualquer modo de endereçamento indexado em alguns computadores.

FIGURA A.6 Seleção dos modos de endereçamento com exemplos, significado e uso.
Nos modos de endereçamento escalonado e de autoincremento/autodecremento, a variável *d* indica o tamanho do item de dados sendo acessado (ou seja, se a instrução está acessando 1, 2, 4 ou 8 bytes). Esses modos de endereçamento são úteis apenas quando os elementos que estão sendo acessados são adjacentes na memória. Os computadores RISC usam endereçamento de deslocamento para simular indireto de registrador com 0 como o endereço e para simular endereçamento direto usando 0 no registrador de base. Em nossas medições, usamos o primeiro nome mostrado para cada modo. As extensões para C usadas como descrições de hardware são definidas na página A-32.

Nessa figura, apenas um recurso não C é usado: a seta para a esquerda (←) é usada para atribuição. Também usamos o array Mem como o nome para a memória principal e o array Regs para registradores. Portanto, Mem[Regs[R1]] se refere ao conteúdo do local de memória cujo endereço é dado pelo conteúdo do registrador 1 (R1). Mais adiante, introduziremos extensões para acessar e transferir dados menores que uma palavra.

Os modos de endereçamento possuem a capacidade de reduzir significativamente as contagens de instrução; eles também aumentam a complexidade de construir um computador e podem elevar o CPI (ciclos de clock por instrução) médio dos computadores que implementam esses modos. Assim, o uso de vários modos de endereçamento é muito importante para ajudar o projetista a escolher o que incluir.

A Figura A.7 mostra os resultados de medir padrões de uso de modos de endereçamento em três programas na arquitetura VAX. Usamos a velha arquitetura VAX para algumas medições neste apêndice porque ela tem o conjunto mais rico de modos de endereçamento e o mínimo de restrições sobre o endereçamento de memória. Por exemplo, a Figura A.6 na página A-8 mostra todos os modos que o VAX aceita. A maioria das medições neste apêndice, contudo, usará as arquiteturas registrador-registrador mais recentes para mostrar como os programas usam conjuntos de instruções de computadores atuais.

Como mostra a Figura A.7, o endereçamento imediato e o de deslocamento dominam o uso dos modos de endereçamento. Vejamos algumas propriedades desses dois modos amplamente utilizados.

Modo de endereçamento de deslocamento

A principal questão que surge para um modo de endereçamento no estilo de deslocamento é a da faixa de deslocamentos usada. Com base no uso de vários tamanhos de deslocamen-

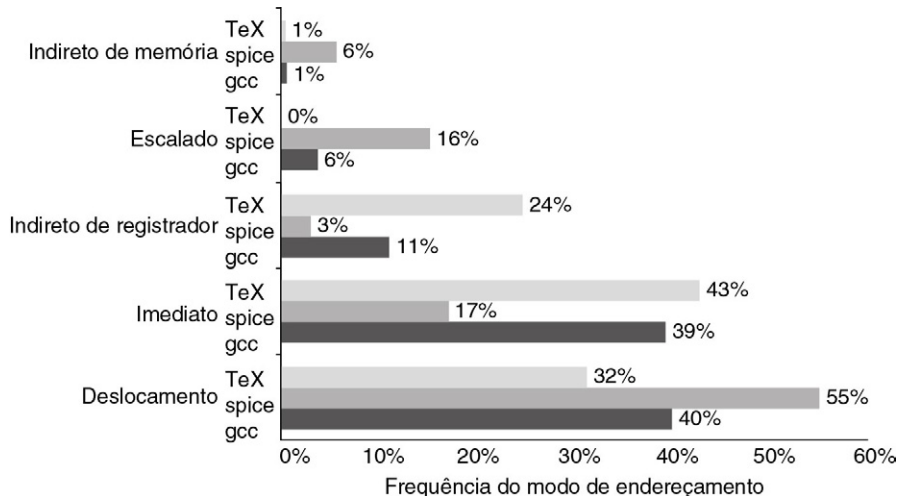



FIGURA A.7 Resumo do uso dos modos de endereçamento de memória (incluindo imediatos).
Esses principais modos de endereçamento são responsáveis por apenas 0-3% dos acessos à memória. Os modos de registrador, que não são contados, respondem por metade das referências de operando, enquanto os modos de endereçamento de memória (incluindo imediato) respondem pela outra metade. Obviamente, o compilador afeta os modos de endereçamento que são usados; veja a Seção A.8. O modo indireto de memória no VAX pode usar deslocamento, autoincremento ou autodecremento para formar o endereço de memória inicial; nesses programas, quase todas as referências indiretas à memória usam o modo de deslocamento como base. O modo de deslocamento inclui todos os tamanhos de deslocamento (8, 16 e 32 bits). Os modos de endereçamento relativos ao PC, usados quase exclusivamente para desvios, não estão incluídos. Apenas os modos de endereçamento com frequência média de mais de 1% são mostrados.

to, uma decisão de que tamanhos serão aceitos pode ser tomada. É importante escolher os tamanhos de campo de deslocamento porque eles afetam diretamente o tamanho da instrução. A Figura A.8 mostra as medições feitas no acesso a dados em uma arquitetura carregamento-armazenamento usando nossos programas de benchmark. Examinamos os offsets de desvio na Seção A.6 — os padrões de acesso a dados e desvios são diferentes; há pouca vantagem em combiná-los, embora, na prática, os tamanhos dos imediatos sejam considerados iguais por simplicidade.

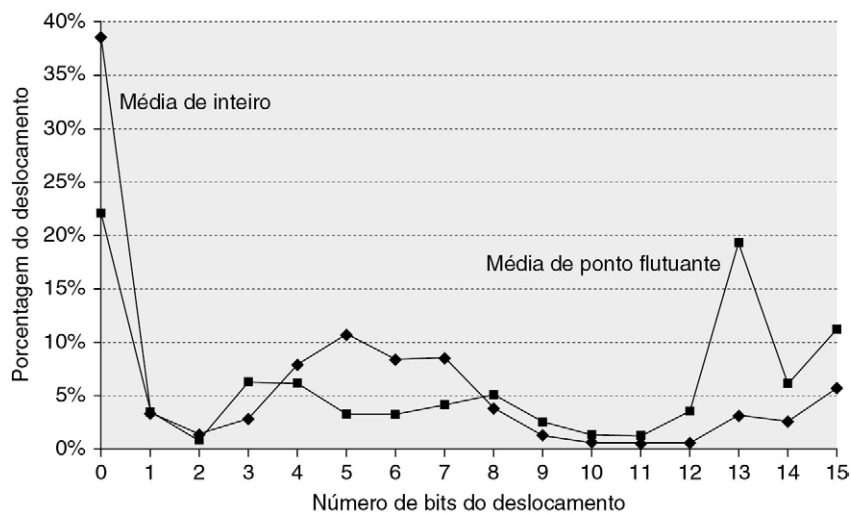



FIGURA A.8 **Os valores de deslocamento são largamente distribuídos.**
Existe grande número de valores pequenos e elevado número de valores grandes. A ampla distribuição dos valores de deslocamento é devida às múltiplas áreas de armazenamento para variáveis e diferentes deslocamentos para acessá-las (Seção A.8), bem como ao esquema de endereçamento geral que o compilador usa. O eixo *x* é $\log_2$ do deslocamento, ou seja, o tamanho de um campo necessário para representar a magnitude do deslocamento. O zero no eixo *x* mostra a porcentagem dos deslocamentos do valor 0. O gráfico não inclui o bit de sinal, que é enormemente afetado pelo leiaute do armazenamento. A maioria dos deslocamentos é positiva, mas uma minoria dos deslocamentos maiores (mais de 14 bits) é negativa. Como esses dados foram coletados em um computador com deslocamentos de 16 bits, eles não podem nos dizer sobre deslocamentos maiores. Esses dados foram tirados em uma arquitetura Alpha com total otimização (Seção A.8) sobre o SPEC CPU2000, mostrando a média dos programas de inteiro (CINT2000) e a média dos programas de ponto flutuante (CFP2000).

Modo de endereçamento imediato ou literal

Os imediatos podem ser usados nas operações aritméticas, em comparações (principalmente para desvios) e em movimentos onde uma constante é necessária em um registrador. O último caso ocorre para constantes escritas no código, que tendem a ser pequenas, e para constantes de endereço, que costumam ser grandes. Para uso dos imediatos, é importante saber se eles precisam ser aceitos para todas as operações ou apenas para um subconjunto. A Figura A.9 mostra a frequência dos imediatos para as classes gerais das operações de inteiro e ponto flutuante em um conjunto de instruções.

Outra importante medição de conjunto de instruções é a faixa de valores para imediatos. Assim como os valores de deslocamento, o tamanho dos valores de imediato afeta o tamanho da instrução. Como mostra a Figura A.10, valores de imediato pequenos são extensamente usados. Imediatos grandes, no entanto, algumas vezes são usados, mais provavelmente em cálculos de endereçamento.

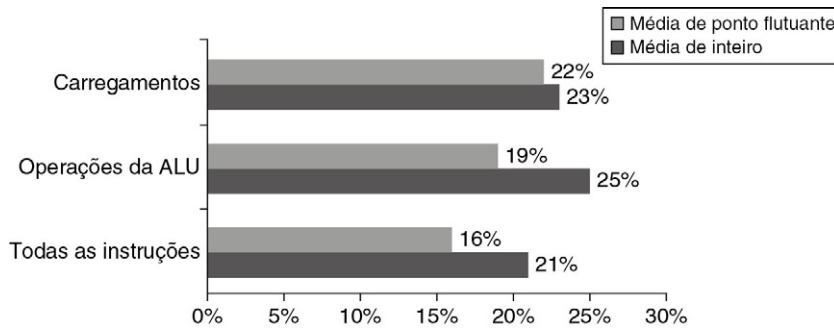



FIGURA A.9 **Aproximadamente 1/4 das transferências de dados e operações da ULA possui um operando imediato.**
As barras inferiores mostram que os programas de inteiro usam imediatos em cerca de 1/5 das instruções, enquanto os programas de ponto flutuante usam imediatos em aproximadamente 1/6 das instruções. Para carregamentos, a instrução de carregamento imediato carrega 16 bits em uma das metades de um registrador de 32 bits. Os carregamentos imediatos não são carregamentos em um sentido estrito porque não acessam a memória. Ocasionalmente, um par de carregamentos imediatos é usado para carregar uma constante de 32 bits, mas isso é raro. (Para operações da ULA, deslocamentos por um valor constante são incluídos como operações com operandos imediatos.) Os programas e o computador usados para coletar essas estatísticas são iguais aos da Figura A.8.

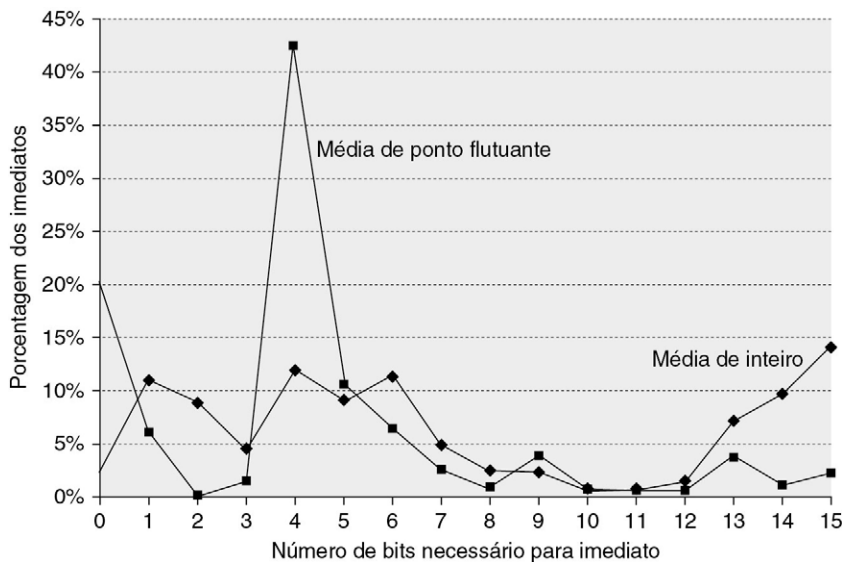



FIGURA A.10 **Distribuição de valores imediatos.**
O eixo *x* mostra o número de bits necessário para representar a magnitude de um valor imediato — 0 significa que o valor do campo imediato era 0. A maioria dos valores imediatos é positiva. Cerca de 20% foram negativos para CINT2000 e aproximadamente 30% foram negativos para CFP2000. Essas medições foram realizadas no Alpha, onde o imediato máximo é 16 bits, para os mesmos programas da Figura A.8. Uma medição semelhante no VAX, que aceitava imediatos de 32 bits, mostrou que cerca de 20-25% dos imediatos eram mais longos que 16 bits. Portanto, 16 bits capturariam aproximadamente 80% e 8 bits cerca de 50%.

Resumo: endereçamento de memória

Primeiro, devido à sua popularidade, esperaríamos que uma nova arquitetura aceitasse pelo menos os seguintes modos de endereçamento: deslocamento, imediato e indireto de registrador. A Figura A.7 mostra que eles representam 75-99% dos modos de endereçamento usados em nossas medições. Segundo, esperaríamos que o tamanho do endereço para o modo de deslocamento fosse pelo menos 12-16 bits, já que a legenda na Figura A.8 sugere que esses tamanhos cobririam 75-99% dos deslocamentos. Terceiro, esperaríamos que o tamanho do campo imediato fosse de pelo menos 8-16 bits, pois cobririam 50-80% dos imediatos.

Tendo coberto as classes de conjunto de instruções e decidido pelas arquiteturas registrador-registrador, além das recomendações anteriores sobre os modos de endereçamento de dados, examinaremos a seguir os tamanhos e as medições dos dados.

A.4 TIPO E TAMANHO DOS OPERANDOS

Como é designado o tipo de um operando? Normalmente, codificar no opcode designa o tipo de operando — esse é o método mais utilizado. Alternativamente, os dados podem ser anotados com tags que são interpretadas pelo hardware. Essas tags especificam o tipo do operando, e a operação é escolhida de acordo. Os computadores com dados identificados por tags, no entanto, só podem ser encontrados em museus.

Vamos começar com as arquiteturas de desktop e de servidores. Em geral, o tipo de operando — inteiro, ponto flutuante de precisão simples, caractere etc. — efetivamente fornece o tamanho. Tipos de operando comuns incluem caractere (8 bits), meia palavra (16 bits), palavra (32 bits), ponto flutuante de precisão simples (também uma palavra) e ponto flutuante de precisão dupla (duas palavras). Inteiros são quase universalmente representados como números binários em complemento de dois. Os caracteres normalmente estão em ASCII, mas o Unicode de 16 bits (usado em Java) está ganhando popularidade com a internacionalização dos computadores. Até o início da década de 1980, a maioria dos fabricantes de computador escolhia sua própria representação de ponto flutuante. Quase todos os computadores a partir daquela época seguiram o mesmo padrão para ponto flutuante, o padrão IEEE 754. O padrão de ponto flutuante IEEE será discutido em detalhes no Apêndice J.

Algumas arquiteturas fornecem operações em strings de caractere, embora essas operações normalmente sejam muito limitadas e tratem cada byte na string como um único caractere. Operações típicas baseadas em strings de caractere são comparações e movimentações.

Para aplicações empresariais, algumas arquiteturas aceitam um formato decimal, normalmente chamado de *decimal compacto* ou *decimal codificado em binário* (*código BCD*) — 4 bits são usados para codificar os valores 0-9 e dois dígitos decimais são compactados em cada byte. As strings de caractere numérico às vezes são chamadas *decimal não compactado*, e as operações — chamadas *compactação* ou *descompactação* — normalmente são fornecidas para a conversão entre elas.

Uma razão para usar operandos decimais é obter resultados que coincidam exatamente com números decimais, já que algumas frações decimais não têm uma representação exata em binário. Por exemplo, $0{,}10_{10}$ é uma fração simples em decimal, mas, em binário, ela requer uma quantidade infinita de dígitos repetitivos: $0{,}00011\overline{0011}\ldots_2$ Assim, cálculos que são exatos em decimal podem estar próximos mas imprecisos em binário, o que pode ser um problema para transações financeiras (veja mais sobre aritmética de precisão no Apêndice J).

Nossos benchmarks SPEC usam tipos de dados de byte ou caractere, meia palavra (inteiro curto), palavra (inteiro), palavra dupla (inteiro longo) e ponto flutuante. A Figura A.11 mostra a distribuição dinâmica dos tamanhos dos objetos referenciados pela memória para esses programas. A frequência de acesso a tipos de dados diferentes ajuda a decidir quais os tipos mais importantes para se ter um suporte eficiente. O computador deve ter um datapath de 64 bits ou seria satisfatório dois ciclos para acessar uma palavra dupla? Como vimos anteriormente, os acessos de byte requerem uma rede de alinhamento: quão importante é aceitar bytes como primitivos? A Figura A.11 usa referências de memória para examinar os tipos de dados que estão sendo acessados.

Em algumas arquiteturas, objetos em registradores podem ser acessados como bytes ou meias palavras. Porém, tal acesso é muito raro — no VAX, isso representa não mais

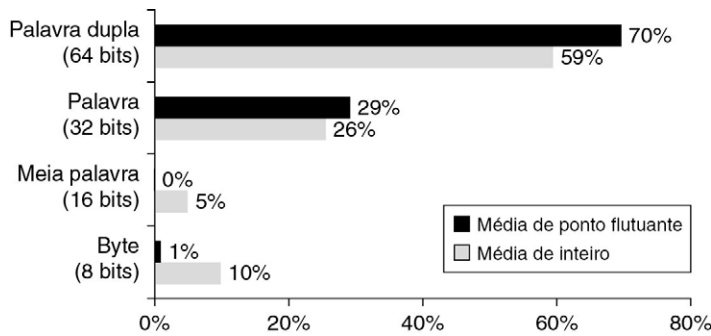



FIGURA A.11 **Distribuição de acessos a dados por tamanho para os programas de benchmark.**
O tipo de dados palavra dupla é usado para ponto flutuante de dupla precisão em programas de ponto flutuante e para endereços, desde que o computador use endereços de 64 bits. Em um computador de endereço de 32 bits, os endereços de 64 bits seriam substituídos por endereços de 32 bits e, portanto, quase todos os endereços de palavra dupla nos programas de inteiro se tornariam endereços de palavra simples.

de 12% das referências de registradores ou aproximadamente 6% de todos os acessos de operandos nesses programas.

A.5 OPERAÇÕES NO CONJUNTO DE INSTRUÇÕES

Os operadores aceitos pela maioria das arquiteturas de conjunto de instruções podem ser categorizados como na Figura A.12. Uma regra geral para todas as arquiteturas é que as instruções mais frequentemente executadas são as operações simples de um conjunto de instruções. Por exemplo, a Figura A.13 mostra 10 instruções simples que respondem por

Tipo de operador	Exemplos
Aritmético e lógico	Operações aritméticas e lógicas de inteiros: adição, subtração e (and), ou (or), multiplicação, divisão
Transferência de dados	Carregamentos-armazenamentos (instruções de movimentação em computadores com endereçamento de memória)
Controle	Desvio, salto, chamada e retorno de procedimento, traps
Sistema	Chamada de sistema operacional, instruções de gerenciamento de memória virtual
Ponto flutuante	Operações de ponto flutuante: adição, multiplicação, divisão, comparação
Decimal	Adição decimal, multiplicação decimal, conversão decimal para caracteres
String	Movimento de string, comparação de string, pesquisa de string
Gráficos	Operações de pixels e vértices, operações de compactação/ descompactação

FIGURA A.12 **Categorias dos operadores de instrução e exemplos de cada uma.**
Todos os computadores normalmente fornecem um conjunto completo de operações para as três primeiras categorias. O suporte para funções de sistema no conjunto de instruções varia amplamante entre as arquiteturas, mas todos os computadores precisam ter algum suporte de instrução para as funções básicas do sistema. A quantidade de suporte no conjunto de instruções para as quatro últimas categorias pode variar de nenhum até um extenso conjunto de instruções especiais. As instruções de ponto flutuante serão fornecidas em qualquer computador destinado a uma aplicação que utiliza intensamente ponto flutuante. Algumas vezes, essas instruções são parte de um conjunto de instruções opcional. As instruções de decimais e string são, às vezes, primitivas, como no VAX ou IBM 360, ou podem ser sintetizadas pelo compilador a partir de instruções mais simples. As instruções gráficas normalmente operam em muitos itens de dados menores em paralelo, por exemplo, realizando oito adições de 8 bits em operandos de 64 bits.

Classificação	Instrução do 80x86	Média de inteiro (% total executada)
1	load	22%
2	desvio condicional	20%
3	compare	16%
4	store	12%
5	add	8%
6	and	6%
7	sub	5%
8	move registrador-registrador	4%
9	call	1%
10	return	1%
Total		**96%**

FIGURA A.13 **As 10 instruções principais para o 80x86.**
As instruções simples dominam essa lista e são responsáveis por 96% das instruções executadas. Essas porcentagens são a média dos cinco programas SPECint92.

96% das instruções executadas para uma coleção de programas de inteiro sendo executados no popular Intel 80x86. Consequentemente, o implementador dessas instruções deve cuidar para torná-las rápidas, já que elas são o caso comum.

Como já dissemos, as instruções na Figura A.13 são encontradas em todo computador para qualquer aplicação — desktop, servidor, embarcado — com as variações das operações na Figura A.12 dependendo fortemente dos tipos de dados tratados pelo conjunto de instruções.

A.6 INSTRUÇÕES PARA FLUXO DE CONTROLE

Como as medidas de comportamento de desvios e saltos são completamente independentes das outras medidas e aplicações, examinamos agora o uso das instruções de fluxo de controle, que têm pouco em comum com as operações das seções anteriores.

Não há uma terminologia consistente para instruções que mudam o fluxo de controle. Na década de 1950, elas normalmente eram chamadas *transferências*. A partir de 1960, o nome *desvio* começou a ser usado. Mais tarde, os computadores introduziram nomes adicionais. Em todo este livro, usamos *salto* quando a mudança no controle for incondicional e *desvio* quando a mudança for condicional.

Podemos distinguir quatro tipos diferentes de mudança de fluxo de controle:

- Desvios condicionais
- Saltos
- Chamadas de procedimento
- Retornos de procedimento

Queremos saber a frequência relativa desses eventos, já que cada evento é diferente, pode usar instruções diferentes e ter comportamento diferente. A Figura A.14 mostra as frequências dessas instruções de fluxo de controle para um computador de carregamento-armazenamento executando nossos benchmarks.

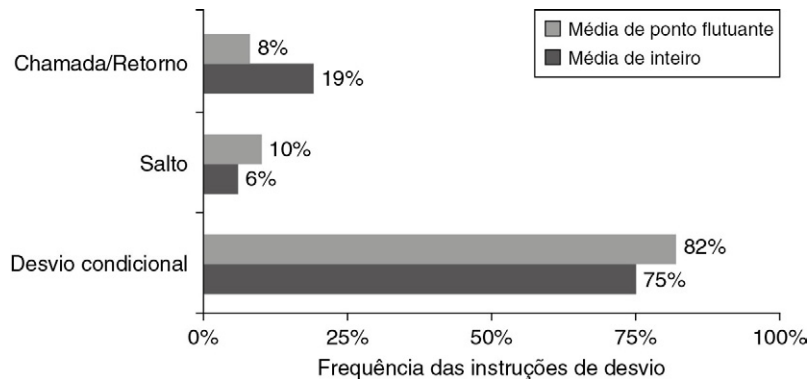



FIGURA A.14 **As instruções de fluxo de controle podem ser divididas em três classes: chamadas ou retornos, saltos e desvios condicionais.**
Os desvios condicionais dominam claramente. Cada tipo é contado em uma de três barras. Os programas e o computador usados para coletar essas estatísticas são os mesmos da Figura A.8.

Modos de endereçamento para instruções de fluxo de controle

O endereço de destino de uma instrução de fluxo de controle sempre deve ser especificado. Esse destino é especificado explicitamente na instrução, na grande maioria dos casos — sendo o retorno de procedimento a principal exceção, já que, para o retorno, o destino não é conhecido em tempo de compilação. O modo mais comum de especificar o destino é fornecer um deslocamento que seja acrescentado ao *contador de programa* (PC). As instruções de fluxo de controle desse tipo são chamadas *relativas ao PC*. Os desvios ou saltos relativos ao PC são vantajosos porque, frequentemente, o destino está próximo à instrução atual, e especificar a posição relativa ao PC atual requer menos bits. Usar o endereçamento relativo ao PC também permite que o código seja executado independentemente de onde esteja carregado. Essa propriedade, chamada *independência de posição*, pode eliminar algum trabalho quando o programa é linkeditado e também é útil em programas linkeditados dinamicamente durante a execução.

Para implementar retornos e saltos indiretos quando o destino não é conhecido em tempo de compilação, é necessário um método diferente do endereçamento relativo ao PC. Aqui deve haver uma maneira de especificar o destino dinamicamente, para que ele possa mudar em tempo de execução. Esse endereço dinâmico pode ser tão simples quanto nomear um registrador que contém o endereço de destino; alternativamente, o salto pode permitir que qualquer modo de endereçamento seja usado para fornecer o endereço de destino.

Esses saltos indiretos de registrador também são úteis para outros quatro recursos importantes:

- Instruções *case* ou *switch*, encontradas na maioria das linguagens de programação (que selecionam uma entre várias alternativas).
- *Funções* ou *métodos virtuais* em linguagens orientadas a objeto, como C++ ou Java (que permitem que diferentes rotinas sejam chamadas dependendo do tipo de argumento).
- *Funções de ordem superior* ou *ponteiros de função* em linguagens como C ou C++ (que permitem que funções sejam passadas como argumentos, dando um pouco do "sabor" da programação orientada a objeto).
- *Bibliotecas compartilhadas dinamicamente* (que permitem carregar e linkeditar uma biblioteca em tempo de execução apenas quando é efetivamente chamada pelo programa em vez de carregada e linkeditada estaticamente antes de o programa ser executado).

Nos quatro casos, o endereço de destino não é conhecido em tempo de compilação e, consequentemente, costuma ser carregado da memória em um registrador antes do salto indireto de registrador.

Como os desvios geralmente usam endereçamento relativo ao PC para especificar seus destinos, uma importante questão é quanto os destinos de desvio estão distantes dos desvios. Saber a distribuição desses deslocamentos ajudará na escolha dos deslocamentos (offsets) de desvio que devem ser suportados e, assim, afetará o tamanho das instruções e a codificação. A Figura A.15 mostra a distribuição dos deslocamentos para desvios relativos ao PC nas instruções. Aproximadamente 75% dos desvios estão voltados para a frente.

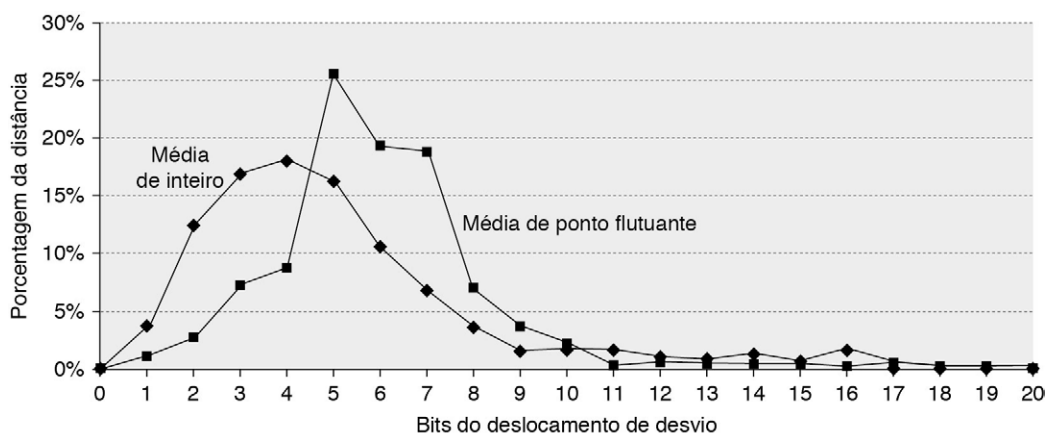



FIGURA A.15 **Distâncias de desvio em termos do número de instruções entre o destino e a instrução de desvio.**

Os desvios mais frequentes nos programas de inteiro são para destinos que podem ser codificados em 4-8 bits. Esse resultado nos diz que campos de deslocamento curtos normalmente são suficientes para desvios e que o projetista pode ganhar alguma densidade de codificação tendo uma instrução mais curta com um deslocamento de desvio menor. Essas medições foram feitas em um computador de carregamento-armazenamento (arquitetura Alpha) com todas as instruções alinhadas nos limites da palavra. Uma arquitetura que exige menos instruções para o mesmo programa, como um VAX, teria distâncias de desvio mais curtas. Entretanto, o número de bits necessários para o deslocamento pode aumentar se o computador tiver instruções de tamanho variável para serem alinhadas em qualquer limite de byte. Os programas e o computador usados para coletar essas estatísticas são os mesmos da Figura A.8.

Opções do desvio condicional

Como a maioria das mudanças no fluxo ocorre em desvios, é importante decidir como especificar a condição do desvio. A Figura A.16 mostra as três principais técnicas em uso atualmente e suas vantagens e desvantagens.

Uma das propriedades mais notáveis dos desvios é que um grande número de comparações consiste em testes simples e outro grande número em comparações com zero. Assim, algumas arquiteturas escolhem tratar essas comparações como casos especiais, sobretudo se uma instrução *compare e desvio* estiver sendo usada. A Figura A.17 mostra a frequência das diferentes comparações usadas para desvio condicional.

Opções de chamada de procedimento

Chamadas e retornos de procedimento incluem transferência de controle e, possivelmente, a gravação de algum estado; no mínimo, o endereço de retorno deve ser salvo em algum lugar, às vezes em um registrador de link especial ou simplesmente em um GPR. Algumas

Nome	Exemplos	Como a condição é testada	Vantagens	Desvantagens
Código de condição (CC)	80x86, ARM, PowerPC, SPARC, SuperH	Testa bits especiais definidos pelas operações de ULA, possivelmente sob o controle do programa.	Algumas vezes, a condição é definida sem custo.	CC é um estado extra. Os códigos de condição restringem a ordenação das instruções, já que eles passam informações de uma instrução para um desvio.
Registrador de condição	Alpha, MIPS	Testa um registrador arbitrário com o resultado de uma comparação.	Simples.	Consome um registrador.
Comparação e desvio	PA-RISC, VAX	A comparação é parte do desvio. Muitas vezes, a comparação é limitada a um subconjunto.	Uma instrução em vez de trabalho por duas para um desvio.	Pode envolver muito trabalho por instrução para execução em pipeline.

FIGURA A.16 Os principais métodos para avaliar condições de desvio, suas vantagens e suas desvantagens.
Embora os códigos de condição possam ser definidos por operações da ULA necessárias para outros fins, as medições nos programas mostram que isso raramente acontece. Os principais problemas de implementação com códigos de condição surgem quando o código de condição é definido por um subconjunto de instruções grande ou aleatoriamente escolhido, em vez de ser controlado por um bit na instrução. Os computadores com comparação e desvio normalmente limitam o conjunto de comparações e usam um registrador de condição para comparações mais complexas. Muitas vezes, técnicas diferentes são usadas para desvios com base em comparação de ponto flutuante em vez daquelas baseadas em comparação de inteiro. Essa dicotomia faz sentido, já que o número de desvios que dependem das comparações de ponto flutuante é muito menor do que o número dependendo das comparações de inteiro.

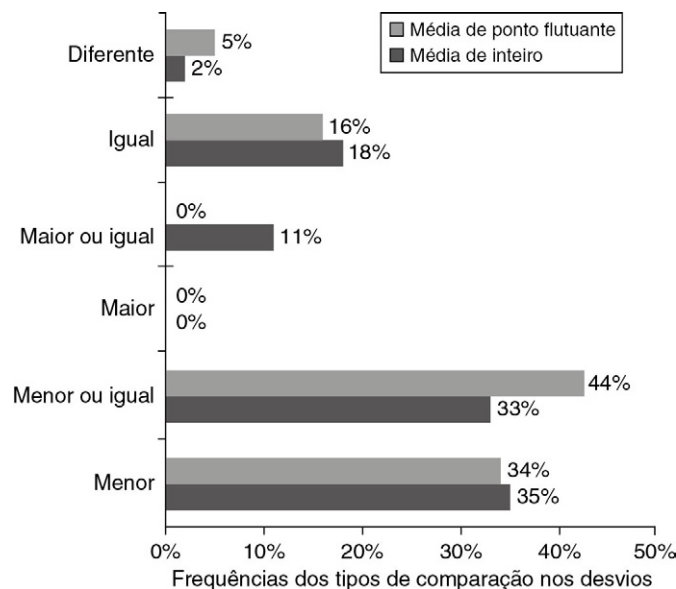



FIGURA A.17 Frequência de diferentes tipos de comparações em desvios condicionais.
Desvios menores (ou iguais) dominam essa combinação de compilador e arquitetura. Essas medições incluem as comparações de inteiros e ponto flutuante nos desvios. Os programas e o computador utilizados para coletar essas estatísticas são iguais aos da Figura A.8.

arquiteturas mais antigas fornecem um mecanismo para salvar muitos registradores, enquanto arquiteturas mais novas exigem que o compilador gere operações de armazenamentos e carregamentos para cada registrador salvo e restaurado.

Há duas convenções básicas em uso para salvar registradores: ou no local de chamada ou dentro do procedimento que está sendo chamado (invocado). *Salvamento por quem chama* significa que o procedimento que chama precisa salvar os registradores que deseja

preservar para o acesso depois da chamada; assim, o procedimento chamado não precisa se preocupar com registradores. *Salvamento pelo chamado* é o contrário: o procedimento chamado precisa salvar os registradores que deseja usar, deixando quem chama livre. Há ocasiões em que o salvamento de quem chama precisa ser usado devido a padrões de acesso a variáveis visíveis globalmente em dois procedimentos diferentes. Por exemplo, suponha que tenhamos um procedimento P1 que chama o procedimento P2 e ambos os procedimentos manipulem a variável global *x*. Se P1 tiver alocado *x* para um registrador, ele deve se certificar em salvar *x* em um local conhecido por P2 antes da chamada a P2. A habilidade de um compilador para descobrir quando um procedimento chamado pode acessar quantidades alocadas em registradores é complicada pela possibilidade da compilação separada. Suponha que P2 não possa acessar *x* mas possa chamar outro procedimento, P3, que pode acessar *x*, embora P2 e P3 sejam compilados separadamente. Devido a essas complicações, a maioria dos compiladores salvará de modo conservador *qualquer* variável que possa ser acessada durante uma chamada.

Nos casos em que qualquer convenção poderia ser usada, alguns programas serão mais eficientes com salvamento pelo chamado e outros serão mais eficientes com salvamento pelo chamador. Como resultado, a maioria dos sistemas reais de hoje usa uma combinação dos dois mecanismos. Essa convenção é especificada em uma interface de aplicação binária (ABI) que estabelece as regras básicas quanto aos registradores que devem ser salvos pelo chamador e quais devem ser salvos pelo chamado. Mais adiante, neste apêndice, examinaremos a divergência entre instruções sofisticadas para salvar automaticamente registradores e as necessidades do compilador.

Resumo: instruções para fluxo de controle

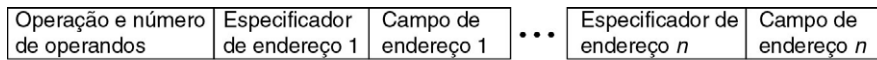
As instruções de fluxo de controle são algumas das instruções mais executadas. Embora haja muitas opções para desvios condicionais, esperávamos que o endereçamento de desvio em uma nova arquitetura fosse capaz de saltar centenas de instruções acima ou abaixo do desvio. Essa exigência sugere um deslocamento de desvio relativo ao PC de pelo menos 8 bits. Esperávamos, também, ver o endereçamento indireto de registrador e o relativo ao PC para instruções de salto, a fim de aceitar retornos, bem como muitos outros recursos dos sistemas atuais.

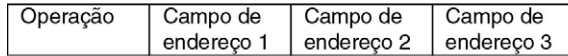
Agora completamos nosso "passeio" pela arquitetura da instrução no nível visto por um programador de linguagem assembly ou um projetista de compilador. Estamos nos voltando para uma arquitetura de carregamento-armazenamento com os modos de endereçamento de deslocamento, imediato e indireto de registrador. Esses dados são inteiros de 8, 16, 32 e 64 bits e dados de ponto flutuante de 32 e 64 bits. As instruções incluem operações simples, desvios condicionais relativos ao PC, instruções de salto e link para chamada de procedimento e saltos indiretos para retorno de procedimento (além de alguns outros usos).

Agora, precisamos escolher como representar essa arquitetura de uma forma que facilite a execução pelo hardware.

A.7 CODIFICAÇÃO DE UM CONJUNTO DE INSTRUÇÕES

Obviamente, as escolhas mencionadas anteriormente afetarão o modo como as instruções são codificadas em uma representação binária para execução pelo processador. Essa representação afeta não só o tamanho do programa compilado, mas também a implementação do processador, que precisa decodificá-la para encontrar rapidamente a operação e seus operandos. A operação normalmente é especificada em um campo

Operação e número de operandos	Especificador de endereço 1	Campo de endereço 1	...	Especificador de endereço *n*	Campo de endereço *n*

(a) Variável (p. ex., Intel 80x86, VAX)

Operação	Campo de endereço 1	Campo de endereço 2	Campo de endereço 3

(b) Fixo (p. ex., Alpha, ARM, MIPS, PowerPC, SPARC, SuperH)

Operação	Especificador de endereço	Campo de endereço

Operação	Especificador de endereço 1	Especificador de endereço 2	Campo de endereço

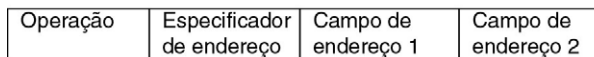
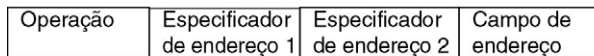
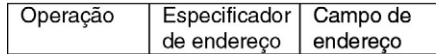
Operação	Especificador de endereço	Campo de endereço 1	Campo de endereço 2

(c) Híbrido (p. ex., IBM 360/370, MIPS16, Thumb, TI TMS320C54x)

FIGURA A.18 **Três variações básicas na codificação de instrução. Tamanho variável, tamanho fixo e híbrido.** O formato variável pode aceitar qualquer número de operandos, com cada especificador de endereço determinando o modo de endereçamento e o tamanho do especificador para esse operando. Ele geralmente permite a menor representação de código, já que os campos não usados não precisam ser incluídos. O formato fixo sempre tem o mesmo número de operandos, com os modos de endereçamento (se houver opções) especificados como parte do opcode. Normalmente, ele resulta no tamanho de código maior. Embora os campos não costumem variar em seu local, eles serão usados para diferentes finalidades por diferentes instruções. O método híbrido possui vários formatos especificados pelo opcode, acrescentando um ou dois campos para especificar o modo de endereçamento e um ou dois campos para especificar o endereço do operando.

chamado *opcode*. Como veremos, uma decisão importante é como codificar os modos de endereçamento com as operações.

Essa decisão depende da faixa dos modos de endereçamento e do grau de independência entre opcodes e modos. Alguns computadores mais antigos possuem 1-5 operandos com 10 modos de endereçamento para cada operando (Fig. A.6). Para um número tão grande de combinações, normalmente um *especificador de endereço* separado é necessário para cada operando: o especificador de endereço informa que modo de endereçamento é usado para acessar o operando. No outro extremo estão computadores de carregamento-armazenamento com um único operando de memória e apenas um ou dois modos de endereçamento; obviamente, nesse caso, o modo de endereçamento pode ser codificado como parte do opcode.

Ao codificar as instruções, o número de registradores e o número de modos de endereçamento têm um impacto significativo no tamanho das instruções, já que o campo de registrador e o campo de modo de endereçamento podem aparecer muitas vezes em uma única instrução. Na realidade, para a maioria das instruções, muito mais bits são consumidos nos campos modo de endereçamento e de registrador do que na especificação do opcode. O arquiteto precisa equilibrar várias forças concorrentes ao codificar o conjunto de instruções:

1. O desejo de ter o máximo de registradores e modos de endereçamento possíveis.
2. O impacto do tamanho dos campos de registrador e do modo de endereçamento no tamanho médio da instrução e, consequentemente, no tamanho médio do programa.
3. O desejo de ter instruções codificadas em tamanhos que sejam fáceis de controlar em uma implementação com pipeline (a importância das instruções facilmente

decodificadas será discutida no Apêndice C e no Capítulo 3). Como um mínimo, o arquiteto quer que as instruções sejam em múltiplos de bytes, e não em um comprimento em bits arbitrário. Muitos arquitetos de computadores desktop e de servidores escolhem usar uma instrução de tamanho fixo para ganhar vantagens de implementação enquanto sacrificam o tamanho médio do código.

A Figura A.18 mostra três escolhas comuns para codificar o conjunto de instruções. À primeira chamamos *variável*, já que permite que quase todos os modos de endereçamento estejam com todas as operações. Esse estilo é melhor quando há muitos modos de endereçamento e operações. À segunda opção, chamamos *fixa*, uma vez que combina a operação e o modo de endereçamento no opcode. Frequentemente, a codificação fixa terá um único tamanho para todas as instruções; ela funciona melhor quando há poucos modos de endereçamento e operações. O compromisso entre codificação variável e codificação fixa é o do tamanho dos programas contra a facilidade de decodificação no processador. A variável tenta usar o mínimo possível de bits para representar o programa, mas as instruções individuais podem variar amplamente em tamanho e na quantidade de trabalho a ser executado.

Vejamos uma instrução do 80x86 para ter um exemplo da codificação variável:

```
add EAX, 1000(EBX)
```

O nome `add` significa uma instrução add de inteiro de 32 bits com dois operandos, e esse opcode leva 1 byte. Um especificador de endereço do 80x86 é de 1 ou 2 bytes, especificando o registrador origem/destino (`EAX`) e o modo de endereçamento (deslocamento nesse caso) e o registrador-base (`EBX`) para o segundo operando. Essa combinação leva 1 byte para especificar os operandos. Quando no modo de 32 bits (Apêndice K), o tamanho do campo endereço é 1 byte ou 4 bytes. Como 1.000 é maior que 2^8, o tamanho total da instrução é

$$1 + 1 + 4 = 6 \text{ bytes}$$

O tamanho das instruções 80x86 varia entre 1-17 bytes. Os programas 80x86 normalmente são maiores do que as arquiteturas RISC, que usam formatos fixos (Apêndice K).

Dados esses dois polos do projeto do conjunto de instruções da variável e fixa, a terceira alternativa imediatamente vem à mente: reduzir a variabilidade em tamanho e trabalho da arquitetura variável, mas fornecer vários tamanhos de instrução para reduzir o tamanho do código. Esse método *híbrido* é a terceira alternativa de codificação, e veremos exemplos em breve.

Tamanho de código reduzido nos RISCs

Quando os computadores RISC começaram a ser usados em aplicações embarcadas, o formato fixo de 32 bits tornou-se um problema, pois o custo e, consequentemente, o tamanho menor do código são importantes. Em resposta, vários fabricantes ofereceram uma nova versão híbrida de seus conjuntos de instruções RISC, com instruções de 16 bits e 32 bits. As instruções curtas aceitam menos operações, menos campos de endereços e campos imediatos, menor quantidade de registradores e um formato de dois endereços em vez do clássico formato de três endereços dos computadores RISC. O Apêndice K fornece dois exemplos, o ARM Thumb e o MIPS16 do MIPS, que ostentam uma redução no tamanho do código de até 40%.

Em contraste com essas extensões de conjunto de instruções, a IBM simplesmente compacta seu conjunto de instruções-padrão e, depois, acrescenta hardware para descompactar ins-

truções, como se elas fossem buscadas da memória em uma falta na cache (miss cache) de instruções. Portanto, a cache de instruções contém instruções de 32 bits completas, mas o código compactado é mantido na memória principal, em ROM e no disco. A vantagem do MIPS16 e do Thumb é que as caches de instrução agem como se fossem aproximadamente 25% maiores, enquanto o CodePack da IBM significa que os compiladores não precisam ser alterados para controlar diferentes conjuntos de instruções, e a decodificação da instrução pode permanecer simples.

O CodePack começa com compactação de codificação run-length em qualquer programa PowerPC e, depois, carrega as tabelas de compactação resultantes em uma tabela de 2 KB no chip. Consequentemente, todo programa tem sua própria codificação única. Para manipular desvios, que não são mais para um limite de palavra, o PowerPC cria uma tabela de hash na memória que faz o mapeamento entre endereços compactados e descompactados. Como um TLB (Cap. 2), ele coloca na cache os mapas de endereço mais recentemente usados, para reduzir o número de acessos à memória. A IBM alega um custo de desempenho geral de 10%, resultando em uma redução de tamanho de código de 35-40%.

A Hitachi simplesmente inventou um conjunto de instruções RISC com formato fixo de 16 bits, chamado SuperH, para aplicações embarcadas (Apêndice H). Ele possui 16 em vez de 32 registradores, para fazê-lo caber no formato mais estreito e com menos instruções, mas, fora isso, se parece com uma arquitetura RISC clássica.

Resumo: codificando um conjunto de instruções

As decisões tomadas nos componentes do projeto de conjunto de instruções, discutidas nas seções anteriores, determinam se o arquiteto tem a escolha entre codificações de instruções variáveis e fixas. Dada a escolha, o arquiteto mais interessado no tamanho de código do que no desempenho escolherá a codificação variável, e o mais interessado no desempenho do que no tamanho de código escolherá a codificação fixa. O Apêndice E traz 13 exemplos dos resultados das escolhas de arquitetos. No Apêndice C e no Capítulo 3, o impacto da variabilidade sobre o desempenho do processador será discutido em detalhes.

Estamos quase terminando de preparar a base para a arquitetura de conjunto de instruções MIPS que será introduzida na Seção A.9. Antes de disso, porém, será útil dar uma olhada na tecnologia do compilador e seu efeito nas propriedades do programa.

A.8 QUESTÕES GERAIS: O PAPEL DOS COMPILADORES

Hoje, quase toda programação é feita em linguagens de alto nível para aplicações de desktops e servidores. Esse desenvolvimento significa que, como a maioria das instruções executadas é a saída de um compilador, uma arquitetura de conjunto de instruções é essencialmente o destino de um compilador. Antigamente, para essas aplicações, muitas vezes eram tomadas decisões arquitetônicas para facilitar a programação em linguagem assembly ou para um kernel específico. Como o compilador afetará significativamente o desempenho de um computador, entender a tecnologia de compilador moderna é vital para projetar e implementar eficientemente um conjunto de instruções.

Já foi comum tentar isolar a tecnologia de compiladores e seu efeito sobre o desempenho de hardware, da arquitetura e seu desempenho, da mesma maneira que era comum tentar separar a arquitetura da sua implementação. Essa separação é praticamente impossível com os compiladores e computadores desktops atuais. As escolhas arquitetônicas afetam positiva ou negativamente a qualidade do código que pode ser gerado para um computador e a complexidade de construir um bom compilador para ele.

Nesta seção, discutiremos os principais objetivos no conjunto de instruções, sobretudo do ponto de vista do compilador. Ela começa com uma revisão da anatomia dos compiladores atuais. Em seguida, discutiremos como a tecnologia do compilador afeta as decisões do arquiteto e como o arquiteto pode dificultar ou facilitar a produção de um bom código pelo compilador. Concluiremos com uma revisão dos compiladores e operações de multimídia, o que, infelizmente, é um mau exemplo de cooperação entre os projetistas de compilador e arquitetos.

A estrutura dos compiladores modernos

Para começar, vejamos como são os compiladores de otimização de hoje. A Figura A.19 mostra a estrutura de compiladores recentes.

O primeiro objetivo de um projetista de compilador é a correção — todos os programas válidos precisam ser compilados corretamente. O segundo objetivo normalmente é a velocidade do código compilado. Em geral, todo um conjunto de outros objetivos segue esses dois, incluindo compilação rápida, suporte à depuração e interoperabilidade entre linguagens. Normalmente, os passos no compilador transformam as representações de alto nível e mais abstratas em representações de nível progressivamente mais baixos. Eventualmente, ele atinge o conjunto de instruções. Essa estrutura ajuda a gerenciar a complexidade das transformações e facilita a escrever um compilador livre de bugs.

A complexidade de escrever um compilador correto é uma grande limitação sobre a quantidade de otimização que pode ser feita. Embora a estrutura de passos múltiplos ajude a reduzir a complexidade do compilador, isso também significa que o compilador precisa ordenar e realizar algumas transformações antes de outras. No diagrama do compilador otimizado na Figura A.19, podemos ver que certas otimizações de alto nível são realizadas

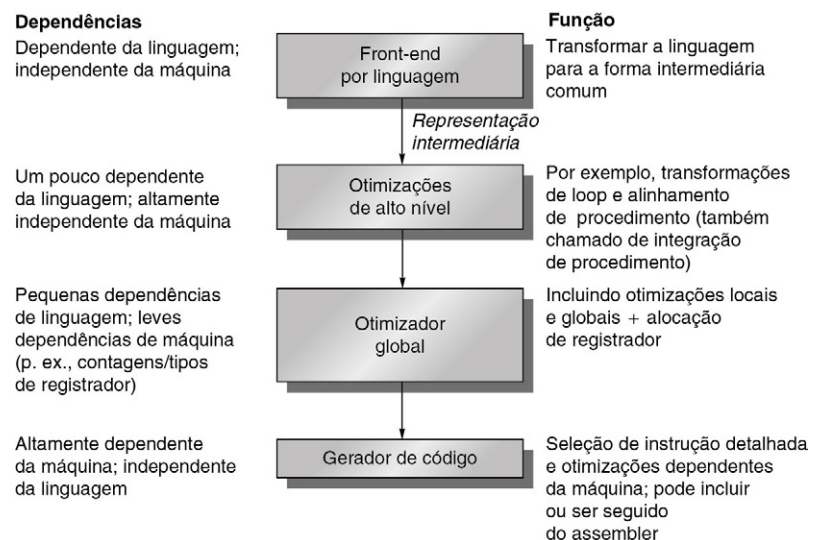



FIGURA A.19 **Os compiladores normalmente consistem em 2-4 passos, com os compiladores de otimização mais alta tendo mais passos.**

Essa estrutura maximiza a probabilidade de um programa compilado em vários níveis de otimização produzir a mesma saída quando recebe a mesma entrada. Os passos de otimização se destinam a ser opcionais e podem ser ignorados quando o objetivo é a compilação mais rápida e o código de qualidade inferior é aceitável. Um passo é simplesmente uma fase em que o compilador lê e transforma o programa inteiro. (O termo *fase* normalmente é usado de maneira indistinta de *passo*.) Como os passos são separados, várias linguagens podem usar os mesmos passos de otimização e de geração código. Apenas um novo front-end é necessário para uma nova linguagem.

muito antes de se saber como será o código resultante. Quando essa transformação é feita, o compilador não pode deixar de voltar e revisitar todas as etapas, possivelmente desfazendo transformações. Essa interação seria proibitiva, tanto em tempo de compilação quanto em complexidade. Portanto, os compiladores fazem suposições sobre a capacidade de etapas futuras lidarem com certos problemas. Por exemplo, os compiladores normalmente precisam escolher quais chamadas de procedimento expandir alinhadas antes de saberem o tamanho exato do procedimento que está sendo chamado. Os projetistas de compilador chamam esse problema de *problema de ordenação de fases*.

Como essa ordenação de transformações interage com a arquitetura do conjunto de instruções? Um bom exemplo ocorre com a otimização chamada *eliminação global de subexpressão comum*. Essa otimização encontra duas instâncias de uma expressão que calculam o mesmo valor e salvam o valor do primeiro cálculo em um temporário. Depois, ela usa o valor temporário, eliminando a segunda computação da expressão comum.

Para que essa otimização seja significante, o temporário deve ser alocado em um registrador. Caso contrário, o custo de armazenar o temporário na memória e depois recarregá-lo pode minimizar as economias obtidas por não recalcular a expressão. Na verdade, há casos em que essa otimização efetivamente torna o código mais lento, quando o temporário não é alocado em um registrador. A ordenação de fase complica esse problema, porque a alocação de registradores normalmente é feita próximo do fim da passagem de otimização global, imediatamente antes da geração do código. Assim, um otimizador que executa essa otimização *deve* supor que o alocador de registradores vai alocar o temporário em um registrador.

As otimizações executadas por compiladores modernos podem ser classificadas pelo estilo da transformação, como segue:

- As *otimizações de alto nível* normalmente são feitas no código-fonte, com a saída como entrada de passos de otimização posteriores.
- As *otimizações locais* só otimizam código dentro de um fragmento de código em linha reta (chamado *bloco básico* pelo pessoal de compiladores).
- As *otimizações globais* estendem as otimizações locais através de desvios e introduzem um conjunto de transformações que visam à otimização de loops.
- A *alocação de registradores* associa registradores com operandos.
- As *otimizações dependentes de processador* tentam tirar proveito de conhecimento de arquiteturas específicas.

Alocação de registradores

Devido ao importante papel desempenhado pela alocação de registradores, tanto em acelerar o código quanto em tornar outras otimizações úteis, ela é uma das mais importantes — se não a mais importante — das otimizações. Os algoritmos de alocação de registradores atuais são baseados em uma técnica chamada *coloração de grafo*. A ideia básica por trás da coloração de grafo é construir um grafo que represente os possíveis candidatos para alocação a um registrador e, depois, usar o grafo para alocar registradores. Geralmente, o problema é como usar um conjunto limitado de cores de modo que nenhum par de nós adjacentes em um grafo de dependência tenha a mesma cor. A ênfase do método é alcançar 100% de alocação das variáveis ativas a registradores. O problema de colorir um grafo em geral pode levar um tempo exponencial em função do tamanho do grafo (NP completo). Entretanto, existem algoritmos heurísticos que funcionam bem na prática, produzindo alocações aproximadas que são executadas em tempo quase linear.

A coloração de grafo funciona melhor quando há pelo menos 16 (e preferivelmente mais) registradores de propósito geral disponíveis para alocação global de variáveis inteiras e

registradores adicionais para ponto flutuante. Infelizmente, a coloração de grafos não funciona muito bem quando o número de registradores é pequeno, porque os algoritmos heurísticos para colorir o grafo provavelmente falharão.

Impacto das otimizações no desempenho

Às vezes é difícil separar algumas das otimizações mais simples — otimizações locais e dependentes do processador — das transformações feitas no gerador de código. Exemplos de otimizações típicas são fornecidos na Figura A.20. A última coluna da Figura A.20 indica a frequência com que as transformações de otimização listadas foram aplicadas ao programa de origem.

A Figura A.21 mostra o efeito de várias otimizações em instruções executadas para dois programas. Nesse caso, os programas otimizados executaram aproximadamente 25-90%

Nome da otimização	Explicação	Porcentagem do número total das transformações otimizadoras
Alto nível	*No nível de código-fonte ou próximo dele; independente do processador*	
Integração de procedimento	Substitui a chamada de procedimento pelo corpo do procedimento	N.M.
Local	*Dentro do código em linha reta*	
Eliminação da subexpressão comum	Substitui duas instâncias da mesma computação por uma única cópia	18%
Propagação constante	Substitui todas as instâncias de uma variável a que é atribuída uma constante pela constante	22%
Redução da altura da pilha	Rearranja a árvore de expressão para minimizar recursos necessários para a avaliação da expressão	N.M.
Global	*Através de um desvio*	
Eliminação global da subexpressão comum	O mesmo que a eliminação global, mas essa versão estende a todos os desvios	13%
Propagação de cópia	Substitui todas as instâncias de uma variável A a que foi atribuído X (p. ex., A = X) por X	11%
Movimento de código	Remove código de um loop que calcula algum valor a cada iteração do loop	16%
Eliminação da variável de indução	Simplifica/elimina os cálculos de endereçamento de array dentro de loops	2%
Dependente do processador	*Depende do conhecimento do processador*	
Redução de força	Muitos exemplos, como substituir multiplicação por uma sequência de adições e deslocamentos	N.M.
Programação de pipeline	Reordena instruções para melhorar o desempenho do pipeline	N.M.
Otimização do deslocamento de desvio	Escolhe o menor deslocamento de desvio que atinja o destino	N.M.

FIGURA A.20 Principais tipos de otimizações e exemplos em cada classe.
Esses dados informam sobre a frequência relativa da ocorrência das várias otimizações. A terceira coluna lista a frequência estática com a qual algumas otimizações comuns são aplicadas em um conjunto de 12 pequenos programas FORTRAN e Pascal. Existem nove otimizações locais e globais feitas pelo compilador incluído na medição. Seis dessas otimizações são cobertas na figura e as outras três respondem por 18% das ocorrências estáticas totais. A abreviatura N.M. significa que o número de ocorrências dessa otimização não foi medido. As otimizações dependentes de processador geralmente são feitas em um gerador de código, e nenhuma delas foi medida nessa experiência. A porcentagem é a parte das otimizações estáticas que são do tipo especificado. *Dados de Chow, 1983 (coletados usando o compilador Stanford UCODE).*

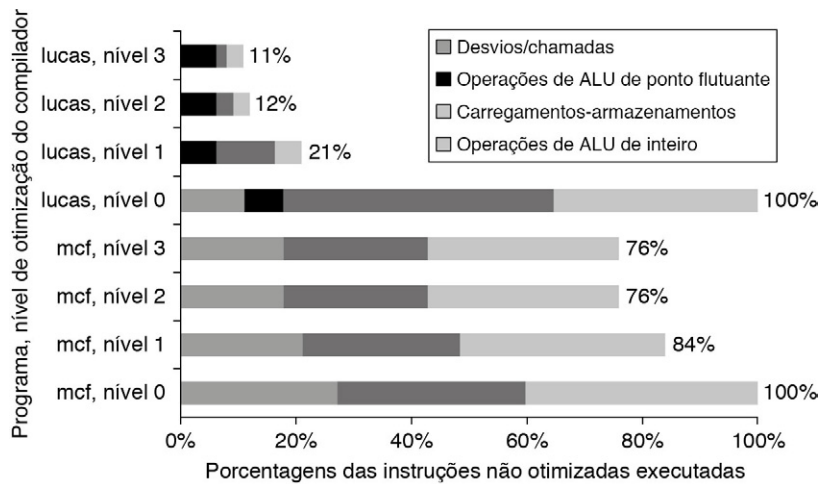



FIGURA A.21 Mudança na contagem de instrução para os programas lucas e mcf do SPEC2000 conforme a variação dos níveis de otimização do compilador.
O nível 0 é igual ao código não otimizado. O nível 1 inclui otimizações locais, programação de código e alocação de registrador local. O nível 2 inclui otimizações globais, transformações de loop (pipelining de software) e alocação de registrador global. O nível 3 acrescenta integração de procedimento. Essas experiências foram realizadas nos compiladores Alpha.

menos instruções do que programas não otimizados. A figura ilustra a importância de olhar o código otimizado antes de sugerir novos recursos de conjuntos de instruções, já que um compilador poderia remover completamente as instruções que o arquiteto estava tentando melhorar.

O impacto da tecnologia de compilador nas decisões do arquiteto

A interação entre os compiladores e as linguagens de alto nível afeta significativamente a maneira como os programas usam a arquitetura de conjunto de instruções. Há duas questões importantes: como as variáveis são alocadas e endereçadas; e quantos registradores são necessários para alocar as variáveis corretamente. Para tratar dessas questões, precisamos examinar três áreas separadas em que as linguagens de alto nível atuais alocam seus dados:

- A *pilha* é usada para alocar variáveis locais. A pilha aumenta ou diminui na chamada ou no retorno de um procedimento, respectivamente. Os objetos na pilha são endereçados em relação ao ponteiro da pilha e são principalmente escalares (variáveis simples) em vez de arrays. A pilha é usada para registros de ativação, *não* como uma pilha para avaliar expressões. Portanto, valores quase nunca são enviados ou recuperados na pilha.
- A *área de dados globais* é usada para alocar objetos estaticamente declarados, como variáveis globais e constantes. Grande parte desses objetos é de arrays e outras estruturas de dados agregadas.
- A *pilha heap* é usada para alocar objetos dinâmicos que não aderem a uma disciplina de pilha. Os objetos na pilha heap são acessados com ponteiros e normalmente não são escalares.

A alocação de registradores é muito mais eficiente para objetos alocados na pilha do que para variáveis globais, e a alocação de registradores é praticamente impossível para objetos alocados na pilha heap, pois eles são acessados com ponteiros. As variáveis globais e algumas variáveis de pilha são impossíveis de alocar, já que usam aliases ou *nomes*

alternativos — existem várias maneiras de se referir ao endereço de uma variável, tornando inválida colocá-la em um registrador. (A maioria das variáveis de heap, na prática, recebe nomes alternativos eficientes para a tecnologia de compilador atual.)

Por exemplo, considere esta sequência de código, em que & retorna o endereço de uma variável e * desreferencia um ponteiro:

```
p = &a-- -- obtém endereço de a em p
a = ...-- atribui diretamente à variável a
*p = ...-- usa p para atribuir à variável a
...a...-- acessa a
```

A variável a não poderia ser alocada a um registrador pela atribuição a *p sem gerar código incorreto. O uso de um nome alternativo causa um grande problema, porque normalmente é difícil ou impossível decidir a quais objetos um ponteiro pode se referir. Um compilador precisa ser conservador; alguns compiladores não alocarão *quaisquer* variáveis locais de um procedimento em um registrador quando houver um ponteiro que possa se referir a *uma* das variáveis locais.

Como o arquiteto pode ajudar o projetista de compiladores

Hoje, a complexidade de um compilador não vem de traduzir declarações simples, como A = B + C. A maioria dos programas é *localmente simples* e as traduções simples funcionam bem. Em vez disso, a complexidade surge porque os programas são grandes e globalmente complexos em suas interações e porque a estrutura dos compiladores significa que, em cada etapa, são tomadas decisões sobre qual é a melhor sequência de código.

Os projetistas de compiladores normalmente estão trabalhando sob seu próprio corolário de um princípio básico na arquitetura: *torne os casos frequentes rápidos e os casos raros corretos*. Ou seja, se soubermos que casos são frequentes e quais deles são raros, e se a geração de código para ambos for direta, então a qualidade do código para o caso raro pode não ser muito importante, mas ele precisa estar correto!

Algumas propriedades de conjunto de instruções ajudam o projetista de compiladores. Essas propriedades não devem ser consideradas regras rígidas, mas orientações que facilitarão a escrita de um compilador que gere um código eficiente e correto.

- *Proporcionar regularidade.* Sempre que fizer sentido, os três componentes primários de um conjunto de instruções — as operações, os tipos de dados e os modos de endereçamento — devem ser *ortogonais*. Dizemos que dois aspectos de uma arquitetura são ortogonais se eles forem independentes. Por exemplo, as operações e os modos de endereçamento são ortogonais se, para cada operação à qual um modo de endereçamento puder ser aplicado, todos os modos de endereçamento forem aplicáveis. Essa regularidade ajuda a simplificar a geração de código e é particularmente importante quando a decisão sobre que código gerar é dividida em duas passagens no compilador. Um bom contraexemplo dessa propriedade é restringir os registradores que podem ser usados para certa classe de instruções. Os compiladores para arquiteturas de registrador de propósito especial normalmente ficam presos nesse dilema. A restrição pode fazer com que o compilador acabe com um monte de registradores disponíveis, mas nenhum do tipo certo!
- *Fornecer primitivas, não soluções.* Os recursos especiais que "correspondem" a uma construção de linguagem ou *a uma* função de kernel normalmente são inúteis. As tentativas de dar suporte a linguagens de alto nível podem funcionar apenas com

uma linguagem ou fazer mais ou menos o que é necessário para uma implementação correta e eficiente da linguagem. Um exemplo de como essas tentativas falharam é dado na Seção A.10.

- *Simplificar os compromissos entre alternativas.* Uma das tarefas mais difíceis de um projetista de compilador é descobrir que sequência de instruções será a melhor para cada segmento de código que surge. Antigamente, as contagens de instruções ou o tamanho de código total podem ter sido boas métricas, mas — como vimos no Capítulo 1 — isso não é mais verdade. Com caches e pipelining, os compromissos se tornaram muito complexos. Qualquer coisa que o arquiteto puder fazer para ajudar o projetista de compilador a entender os custos das sequências alternativas de código ajudará a melhorar o código. Uma das instâncias mais difíceis dos compromissos complexos ocorre na arquitetura de registrador-memória, ao se decidir quantas vezes uma variável deve ser referenciada antes que seja menos oneroso carregá-la em um registrador. É difícil calcular esse limiar; na verdade, pode variar entre modelos da mesma arquitetura.
- *Fornecer instruções que associem as quantidades conhecidas em tempo de compilação como constantes.* Um projetista de compilador detesta a ideia de o processador interpretar, em tempo de execução, um valor que foi conhecido em tempo de compilação. Bons contraexemplos desse princípio incluem instruções que interpretam valores que foram fixados em tempo de compilação. Por exemplo, a chamada de procedimento do VAX (calls) interpreta dinamicamente uma máscara dizendo quais registradores salvar em uma chamada, mas a máscara é fixada em tempo de compilação (Seção A.10).

O suporte do compilador (ou a falta dele) para instruções de multimídia

Infelizmente, os projetistas das instruções SIMD (Seção 4.3, no Cap. 4) basicamente ignoraram a subseção anterior. Essas instruções costumam ser soluções, não primitivas; elas têm poucos registradores; e os tipos de dados não correspondem às linguagens de programação existentes. Os arquitetos esperavam encontrar uma solução barata que ajudasse a alguns usuários, mas, na realidade, apenas algumas rotinas de biblioteca gráfica de baixo nível as utilizam.

As instruções SIMD são, na verdade, uma versão abreviada de um estilo de arquitetura elegante que possui sua própria tecnologia de compilador. Como explicamos na Seção 4.2, as *arquiteturas vetoriais* operam em vetores de dados. Criados originalmente para códigos científicos, em geral os kernels de multimídia são vetorizáveis também, embora frequentemente com vetores menores. Como a Seção 4.3 sugere, podemos pensar no MMX e SSE da Intel ou no PowerPC AltiVec simplesmente como computadores de vetor curto: o MMX, com vetores de oito elementos de 8 bits, quatro elementos de 16 bits ou dois elementos de 32 bits, e o AltiVec, com vetores com o dobro disso. Eles são implementados simplesmente como elementos estreitos adjacentes em grandes registradores.

Essas arquiteturas de microprocessador aumentaram o tamanho de registrador vetorial na arquitetura: a soma dos tamanhos dos elementos é limitada a 64 bits para o MMX e a 128 bits para o AltiVec. Quando a Intel decidiu expandir para vetores de 128 bits, ela acrescentou todo um novo conjunto de instruções, chamado Streaming SIMD Extension (SSE).

Uma grande vantagem dos computadores vetoriais é ocultar a latência do acesso à memória carregando muitos elementos de uma vez e, depois, sobrepondo a execução com transferência de dados. O objetivo dos modos de endereçamento vetoriais é coletar dados espalhados na memória, colocá-los em uma forma compacta para que possam ser operados eficientemente e, depois, colocar os resultados nos lugares a que pertencem.

Os computadores vetoriais tradicionais acrescentaram *endereçamento espaçado* e *endereçamento gather-scatter* (Seção 4.2) para aumentar o número de programas que podem ser vetorizados. O endereçamento espaçado salta um número fixo de palavras entre cada acesso, de modo que o endereçamento sequencial normalmente é chamado *endereçamento espaço único. Gather-scatter* encontra seus endereços em outro registrador vetorial: pense nele como endereçamento indireto de registrador para computadores vetoriais. Por outro lado, sob uma perspectiva vetorial, esses computadores SIMD de vetor curto aceitam apenas acessos com espaçamento unitário: os acessos à memória carregam ou armazenam todos os elementos ao mesmo tempo de um único grande espaço da memória. Como os dados para aplicações de multimídia geralmente são fluxos que começam e terminam na memória, os modos de endereçamento avançado e *gather-scatter* são essenciais para uma vetorização bem-sucedida (Seção 4.7).

Exemplo Como exemplo, compare um computador de vetor com o MMX para conversão de representação de cor de pixels de RGB (vermelho, verde, azul) para YUV (luminosidade, cromaticidade), com cada pixel representado por 3 bytes. A conversão envolve apenas três linhas de código C colocadas em um loop:

```
Y = (9798*R + 19235*G + 3736*B) / 32768;
U = (-4784*R - 9437*G + 4221*B) / 32768 + 128;
V = (20218*R - 16941*G - 3277*B) / 32768 + 128;
```

Um computador vetorial de 64 bits pode calcular 8 pixels simultaneamente. Um computador vetorial para mídia com endereços espaçados usa:

- 3 cargas vetoriais (para obter RGB)
- 3 multiplicações vetoriais (para converter R)
- 6 multiplicações adições vetoriais (para converter G e B)
- 3 deslocamentos vetoriais (para dividir por 32.768)
- 2 adições vetoriais (para somar 128)
- 3 armazenamentos vetoriais (para armazenar YUV)

O total é de 20 instruções para realizar as 20 operações no código C anterior, para converter 8 pixels (Kozyrakis, 2000). (Como um vetor pode ter 32 elementos de 64 bits, esse código, na verdade, converte até $32 \times 8 = 256$ pixels.) Por sua vez, o site da Intel mostra que uma rotina de biblioteca para realizar o mesmo cálculo em 8 pixels leva 116 instruções MMX mais seis instruções 80x86 (Intel, 2001). Esse aumento de seis vezes nas instruções é devido ao grande número de instruções para carregar e desempacotar pixels RGB e para empacotar e armazenar pixels YUV, já que não existem acessos espaçado à memória.

Ter vetores curtos e limitados à arquitetura com poucos registradores e modos de endereçamento de memória simples torna mais difícil usar tecnologia de compilador de vetorização. Assim, essas instruções SIMD têm maior probabilidade de serem encontradas em bibliotecas codificadas à mão do que em código compilado.

Resumo: o papel dos compiladores

Esta seção leva a várias recomendações: primeiro, esperamos que uma nova arquitetura de conjunto de instruções tenha pelo menos 16 registradores de propósito geral — além de registradores separados para números de ponto flutuante — para simplificar a alocação de registradores usando coloração de grafo. O conselho sobre a ortogonalidade sugere que todos os modos de endereçamento aceitos se aplicam a todas as instruções que transferem dados. Finalmente, os três últimos conselhos — fornecer primitivas em vez de soluções, simplificar compromissos entre alternativas e não associar constantes

em tempo de execução — sugerem que é melhor errar pela simplicidade. Em outras palavras, entenda que menos é melhor no projeto de um conjunto de instruções. Infelizmente, as extensões SIMD são mais um exemplo de bom marketing do que um grande sucesso do coprojeto de hardware-software.

A.9 JUNTANDO TUDO: A ARQUITETURA MIPS

Nesta seção, descreveremos uma arquitetura de carregamento-armazenamento (load-store) simples de 64 bits, chamada MIPS. A arquitetura de conjunto de instruções das relativas ao MIPS e ao RISC foi baseada em observações semelhantes às discutidas nas últimas seções (na Seção L.3 discutiremos como e por que essas arquiteturas se tornaram populares). Revendo nossas expectativas de cada seção, para aplicações de desktop:

- *Seção A.2*. Usar registradores de propósito geral com uma arquitetura de carregamento-armazenamento.
- *Seção A.3*. Aceitar esses modos de endereçamento: deslocamento (com tamanho de deslocamento de endereço de 12-16 bits), imediato (tamanho de 8-16 bits) e indireto de registrador.
- *Seção A.4*. Aceitar esses tamanhos e tipos de dados: inteiros de 8, 16, 32 e 64 bits e números de ponto flutuante IEEE 754 de 64 bits.
- *Seção A.5*. Aceitar essas instruções simples, já que elas dominarão o número de instruções executadas: carregamento, armazenamento, adição, subtração, mover registrador-registrador e deslocamento.
- *Seção A.6*. Comparação de igualdade, comparação de desigualdade, comparação de menor que, desvio (com um endereço relativo ao PC de pelo menos objeto), salto, chamada e retorno de procedimento.
- *Seção A.7*. Usar codificação de instrução fixa, se estiver interessado em desempenho, e codificação de instrução variável, se estiver interessado em tamanho de código.
- *Seção A.8*. Fornecer pelo menos 16 registradores de propósito geral, certificar que todos os modos de endereçamento se aplicam a todas as instruções de transferência de dados e objetivar um conjunto mínimo de instruções. Esta seção não aborda programas de ponto flutuante, mas eles normalmente usam registradores de ponto flutuante separados. A justificativa é aumentar o número de registradores sem causar problemas no formato de instrução ou na velocidade do banco de registradores (register file) de propósito geral. Esse compromisso, no entanto, não é ortogonal.

Introduzimos o MIPS mostrando como ele segue essas recomendações. Como a maioria dos computadores atuais, o MIPS enfatiza:

- Um conjunto de instruções de carregamento-armazenamento simples.
- Um projeto que visa à eficiência do pipelining (discutido no Apêndice C), incluindo uma codificação fixa de conjunto de instruções.
- Eficiência como uma meta do compilador.

O MIPS fornece um bom modelo arquitetônico para estudo, não só devido à popularidade desse tipo de processador, mas porque essa é uma arquitetura fácil de entender. Usamos essa arquitetura também no Apêndice C e no Capítulo 3, e ela forma a base para diversos exercícios e projetos de programação.

Desde o primeiro processador MIPS em 1985, tem havido muitas versões do MIPS (Apêndice K). Usaremos um subconjunto do que hoje é chamado MIPS64, que normalmente será abreviado para MIPS, mas o conjunto de instruções completo é encontrado no Apêndice K.

Registradores para MIPS

O MIPS64 possui 32 registradores de propósito geral (GPRs) de 64 bits, denominados R0, R1, ..., R31. Às vezes, os GPRs são chamados *registradores de inteiros*. Além disso, existe um conjunto de 32 registradores de ponto flutuante (FPRs), denominados F0, F1, ..., F31, que podem armazenar 32 valores de precisão simples (32 bits) ou valores de precisão dupla (64 bits). (Quando armazena um número de precisão simples, a outra metade do FPR fica ociosa.) Tanto as operações de ponto flutuante de precisão simples quanto as de precisão dupla são disponibilizadas. O MIPS também inclui instruções que operam em dois operandos de precisão simples em um único registrador de ponto flutuante de 64 bits.

O valor de R0 é sempre 0. Mais adiante, veremos como podemos usar esse registrador para sintetizar uma variedade de operações úteis a partir de um conjunto de instruções simples.

Alguns registradores especiais podem ser transformados de e para os registradores de propósito geral. Um exemplo disso é o registrador de status de ponto flutuante, usado para guardar informações sobre os resultados das operações de ponto flutuante. Também existem instruções para mover entre um FPR e um GPR.

Tipos de dados para MIPS

Os tipos de dados são bytes de 8 bits, meias palavras de 16 bits, palavras de 32 bits, palavras duplas de 64 bits para dados de inteiro, precisão simples de 32 bits e precisão dupla de 64 bits para ponto flutuante. As meias palavras foram acrescentadas porque são encontradas em linguagens como C e são populares em alguns programas, como os sistemas operacionais, preocupados com o tamanho das estruturas de dados. Elas também se tornarão mais comuns se o Unicode se tornar bastante usado. Os operandos de ponto flutuante de precisão simples foram acrescentados por questões semelhantes. (Lembre-se da advertência inicial de que você deve medir muitos outros programas antes de projetar um conjunto de instruções.)

Os operandos do MIPS64 funcionam em inteiros de 64 bits e ponto flutuante de 32 ou 64 bits. Bytes, meias palavras e palavras são carregados nos registradores de propósito geral com zeros ou o bit de sinal replicado para preencher os 64 bits dos GPRs. Uma vez carregados, eles são usados em operações de inteiros de 64 bits.

Modos de endereçamento para transferências de dados MIPS

Os únicos modos de endereçamento de dados são o modo imediato e o de deslocamento, ambos com campos de 16 bits. O modo indireto de registrador é conseguido simplesmente colocando 0 no campo de deslocamento de 16 bits, e o endereçamento absoluto com um campo de 16 bits é realizado usando o registrador 0 como o registrador-base. Usar zero nos dá quatro modos efetivos, embora apenas dois sejam aceitos na arquitetura.

A memória MIPS é endereçável por byte com um endereço de 64 bits. Ela tem um bit de modo que permite ao software selecionar Big Endian ou Little Endian. Por se tratar de uma arquitetura de carregamento-armazenamento (load-store), todas as referências entre a memória e os GPRs ou FPRs são através de carregamentos ou armazenamentos. Aceitando os tipos de dados mencionados, os acessos à memória envolvendo GPRs podem ser para um byte, meia palavra ou palavra dupla. Os FPRs podem ser carregados e armazenados com números de precisão simples ou precisão dupla. Todos os acessos à memória precisam ser alinhados.

Formato de instrução do MIPS

Como o MIPS possui apenas dois modos de endereçamento, eles podem ser codificados no opcode. Seguindo o conselho sobre tornar fácil o pipeline e a decodificação do processador, todas as instruções são de 32 bits com um opcode primário de 6 bits. A Figura A.22 mostra o leiaute da instrução. Esses formatos são simples, embora fornecendo campos de 16 bits para endereçamento de deslocamento, constantes imediatas ou endereços de desvio relativos ao PC.

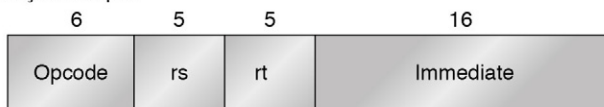
O Apêndice K mostra uma variante do MIPS — chamada MIPS16 — que possui instruções de 16 bits e 32 bits para melhorar a densidade de código para aplicações embarcadas. Ficaremos com o formato de 32 bits tradicional neste livro.

Operações MIPS

O MIPS aceita a lista das operações simples recomendada a seguir, além de algumas outras. Existem quatro grandes classes de instruções: carregamentos e armazenamentos, operações da ULA, desvios e saltos, e operações de ponto flutuante.

Qualquer um dos registradores de propósito geral ou de ponto flutuante pode ser carregado ou armazenado, exceto que o carregamento de R0 não tem qualquer efeito. A Figura A.23 dá exemplos das instruções carregamento e armazenamento. Os números de ponto flutuante de precisão simples ocupam metade de um registrador de ponto flutuante. As conversões entre precisão simples e dupla precisam ser feitas explicitamente. O formato de ponto flutuante é o IEEE 754 (Apêndice J). Uma lista de todas as instruções MIPS em nosso subconjunto aparece na Figura A.26 (página A-35).

Instrução do tipo I

6 5 5 16

Encodes: Carregamentos e armazenamentos de bytes, meias palavras, palavras duplas. Todos os imediatos (RT ← rs op immediate)

Instruções de desvio condicional (rs é registrador, rd não usado)
Registrador de salto, registrador salto e link
(rd = 0, rs = destino, imediato = 0)

Instrução tipo R

6 5 5 5 5 6

Operandos de ALU registrador-registrador: rd ← rs funct rt
Função codifica a operação de caminho de dados: Add, Sub, . . .
Registradores e movimentos especiais de leitura/escrita

Instrução de tipo J

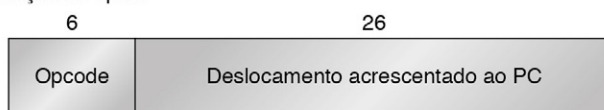
6 26

Jump and jump and link
Trap and return from exception

FIGURA A.22 Leiaute de instrução para MIPS.
Todas as instruções são codificadas em um de três tipos, com campos comuns no mesmo local em cada formato.

Instrução de exemplo	Nome da instrução	Significado
LD R1,30(R2)	Load double word	Regs[R1] $\leftarrow_{64}$ Mem[30+Regs[R2]
LD R1,1000(R0)	Load double word	Regs[R1] $\leftarrow_{64}$ Mem[1000+0]
LW R1,60(R2)	Load word	Regs[R1] $\leftarrow_{64}$(Mem[60+Regs[R2]]0)^{32} ## Mem[60+Regs[R2]]
LB R1,40(R3)	Load byte	Regs[R1] $\leftarrow_{64}$(Mem[40+Regs[R3]]0)^{56} ## Mem[40+Regs[R3]]
LBU R1,40(R3)	Load byte unsigned	Regs[R1] $\leftarrow_{64}$0^{56} ## Mem[40+Regs[R3]]
LH R1,40(R3)	Load half word	Regs[R1] $\leftarrow_{64}$(Mem[40+Regs[R3]] 0)^{48} ## Mem[40+Regs[R3]] ## Mem[41+Regs[R3]]
L.S F0,50(R3)	Load FP single	Regs[F0] $\leftarrow_{64}$ Mem[50+Regs[R3]] ## 0^{32}
L.D F0,50(R2)	Load FP double	Regs[F0] $\leftarrow_{64}$ Mem[50+Regs[R2]]
SD R3,500(R4)	Store double word	Mem[500+Regs[R4]] $\leftarrow_{64}$ Regs[R3]
SW R3,500(R4)	Store word	Mem[500+Regs[R4]] $\leftarrow_{32}$ Regs[R3]$_{32..63}$
S.S F0,40(R3)	Store FP single	Mem[40+Regs[R3]] $\leftarrow_{32}$ Regs[F0]$_{0..31}$
S.D F0,40(R3)	Store FP double	Mem[40+Regs[R3]] $\leftarrow_{64}$ Regs[F0]
SH R3,502(R2)	Store half	Mem[502+Regs[R2]] $\leftarrow_{16}$ Regs[R3]$_{48..63}$
SB R2,41(R3)	Store byte	Mem[41+Regs[R3]] $\leftarrow_{8}$ Regs[R2]$_{56..63}$

FIGURA A.23 As instruções carregamento e armazenamento (load e store) no MIPS.
Todas usam um único modo de endereçamento e exigem que o valor de memória esteja alinhado. Obviamente, tanto os carregamentos quanto os armazenamentos estão disponíveis para todos os tipos de dados mostrados.

Para entender essas figuras, precisamos introduzir algumas extensões adicionais em nossa linguagem de descrição C, usada inicialmente na página A-8:

- Um subscrito é anexado ao símbolo ← sempre que o tamanho do dado que está sendo transferido pode não ser claro. Assim, $\leftarrow_n$ significa uma quantidade de *n* bits. Usamos *x*, *y*←*z* para indicar que *z* deve ser transferido para *x* e *y*.
- Um subscrito é usado para indicar a seleção de um bit de um campo. Os bits são rotulados a partir do bit mais significativo começando em 0. O subscrito pode ser um único dígito (p. ex., Regs[R4]$_0$ produz o bit de sinal R4) ou uma subfaixa (p. ex., Regs[R3]$_{56..63}$ produz o byte menos significativo de R3).
- A variável Mem, usada como um array que representa a memória principal, é indexada por um endereço de byte e pode transferir qualquer número de bytes.
- Um sobrescrito é usado para reproduzir um campo (p. ex., 0^{48} produz um campo de zeros de 48 bits de extensão).
- O símbolo ## é usado para concatenar dois campos e pode aparecer em qualquer lado de uma transferência de dados.

Como exemplo, considerando que R8 e R10 são registradores de 64 bits:

$$\text{Regs[R10]}_{32..63} \leftarrow_{32} (\text{Mem[Regs[R8]]}_0)^{24} \text{ \#\# Mem[Regs[R8]]}$$

significa que o byte no local de memória endereçado pelo conteúdo do registrador R8 é estendido com o sinal para formar uma quantidade de 32 bits, que é armazenada na metade inferior do registrador R10 (a metade superior de R10 fica inalterada).

Todas as instruções da ULA são instruções registrador-registrador. A Figura A.24 fornece alguns exemplos das instruções aritméticas/lógicas. As operações incluem aritmética simples e operações lógicas: adição, subtração, AND, OR, XOR e deslocamentos. As formas imediatas de todas essas instruções são fornecidas usando um imediato de 16 bits es-

Instrução de exemplo	Nome da instrução	Significado
DADDU R1,R2,R3	Add unsigned	Regs[R1] ← Regs[R2]+Regs[R3]
DADDIU R1,R2,#3	Add immediate unsigned	Regs[R1] ← Regs[R2]+3
LUI R1,#42	Load upper immediate	Regs[R1] ← 0^{32}##42##0^{16}
DSLL R1,R2,#5	Shift left logical	Regs[R1] ← Regs[R2]<<5
SLT R1,R2,R3	Set less than	if (Regs[R2]<Regs[R3]) Regs[R1] ← 1 else Regs[R1] ← 0

FIGURA A.24 Exemplos das instruções aritméticas/lógicas no MIPS, com e sem imediatos.

tendido com o sinal. A operação LUI (load upper immediate — carregar imediato superior) carrega os bits 32 a 47 de um registrador, enquanto define o restante do registrador em 0. LUI permite que uma constante de 32 bits seja construída em duas instruções ou uma transferência de dados usando qualquer endereço constante de 32 bits em uma instrução extra.

Como mencionamos, R0 é usado para sintetizar operações populares. Carregar uma constante é simplesmente um add imediato onde o operando de origem é R0, e um move registrador-registrador é simplesmente um add onde uma das origens é R0 (algumas vezes usamos o mnemônico LI, significando load immediate [carregar imediato], para representar o primeiro e o mnemônico MOV para o último).

Instruções de fluxo de controle do MIPS

O MIPS fornece instruções de comparação, que comparam dois registradores para ver se o primeiro é menor que o segundo. Se a condição for verdadeira, essas instruções colocam um 1 no registrador de destino (para representar verdadeiro); caso contrário, elas colocam o valor 0. Como essas operações "definem" um registrador, elas são chamadas set-equal, set-not-equal, set-less-than, e assim por diante. Também há formas imediatas dessas comparações.

O controle é manipulado através de um conjunto de saltos e um conjunto de desvios. A Figura A.25 fornece algumas instruções típicas de desvio e de salto. As quatro instruções de salto são diferenciadas pelas duas maneiras de especificar o endereço de destino e pelo

Instrução de exemplo	Nome da instrução	Significado
J name	Jump	$PC_{36..63}$ ← name
JAL name	Jump and link	Regs[R31] ← PC+8; $PC_{36..63}$ ← name; ((PC+4)-2^{27}) ≤ name < ((PC+4)+2^{27})
JALR R2	Jump and link register	Regs[R31] ← PC+8; PC ← Regs[R2]
JR R3	Jump register	PC ← Regs[R3]
BEQZ R4,name	Branch equal zero	If (Regs[R4]==0) PC ← name; ((PC+4)-2^{17}) ≤ name < ((PC+4)+2^{17})
BNE R3,R4,name	Branch not equal zero	if (Regs[R3]!= Regs[R4]) PC ← name; ((PC+4)-2^{17}) ≤ name < ((PC+4)+2^{17})
MOVZ R1,R2,R3	Conditional move if zero	if (Regs[R3]==0) Regs[R1] ← Regs[R2]

FIGURA A.25 Instruções de fluxo de controle típicas do MIPS.

Todas as instruções de controle, exceto saltos para um endereço em um registrador, são relativas ao PC. Repare que as distâncias de desvio são maiores do que o campo de endereço sugeriria; já que as instruções MIPS são todas com 32 bits, o endereço de desvio de byte é multiplicado por 4 para obter uma distância maior.

fato de um link ser ou não ser feito. Dois saltos usam um offset de 26 bits, deslocado 2 bits e, depois, substituem os 28 bits inferiores do contador de programa (da instrução sequencialmente após o salto) para determinar o endereço de destino. As outras duas instruções de salto especificam um registrador que contém o endereço de destino. Existem dois tipos de saltos: salto simples e salto e link (usado para chamadas de procedimento). Este último coloca o endereço de retorno — o endereço da próxima instrução em sequência — em R31.

Todos os desvios são condicionais. A condição de desvio é especificada pela instrução, que deve testar se o registrador origem é zero ou não zero; o registrador pode conter um valor ou o resultado de uma comparação. Também existem instruções de desvio condicional para testar a igualdade entre dois registradores e se um registrador é negativo. O endereço de destino do desvio é especificado com um deslocamento sinalizado com 16 bits, que é deslocado dois bits à esquerda e, depois, adicionado ao contador de programa (PC), que aponta para a próxima instrução em sequência. Existe também um desvio para testar o registrador de status de ponto flutuante quanto aos desvios condicionais de ponto flutuante, descritos mais adiante.

O Apêndice C e o Capítulo 3 mostram que os desvios condicionais são um grande problema para execução em um pipeline; portanto, muitas arquiteturas acrescentaram instruções para converter um desvio simples em uma instrução aritmética condicional. O MIPS incluiu a movimentação condicional em zero ou não zero. O valor do registrador de destino é deixado inalterado ou é substituído por uma cópia de um dos registradores de origem, dependendo do fato de o valor do outro registrador de origem ser zero ou não.

Operações de ponto flutuante do MIPS

As instruções de ponto flutuante manipulam os registradores de ponto flutuante e indicam se a operação a ser realizada é de precisão simples ou dupla. As operações `MOV.S` e `MOV.D` copiam um registrador de ponto flutuante de precisão simples (`MOV.S`) ou de precisão dupla (`MOV.D`) para outro registrador do mesmo tipo. As operações `MFC1`, `MTC1`, `DMFC1`, `DMTC1` movem dados entre um registrador de ponto flutuante simples ou duplo e um registrador inteiro. As conversões de inteiro para ponto flutuante também são fornecidas e vice-versa.

As operações de ponto flutuante são adição, subtração, multiplicação e divisão; um sufixo `D` é usado para precisão dupla, e um sufixo `S` é usado para precisão simples (p. ex., `ADD.D`, `ADD.S`, `SUB.D`, `SUB.S`, `MUL.D`, `MUL.S`, `DIV.D`, `DIV.S`). As comparações de ponto flutuante definem um bit no registrador especial de status de ponto flutuante que pode ser testado com um par de desvios: `BC1T` e `BC1F`, desvio de ponto flutuante verdadeiro e desvio de ponto flutuante falso.

Para conseguir maior desempenho para rotinas gráficas, o MIPS64 possui instruções que realizam duas operações de ponto flutuante de 32 bits em cada metade do registrador de ponto flutuante de 64 bits. Essas operações *simplesmente emparelhadas* incluem `ADD.PS`, `SUB.PS`, `MUL.PS` e `DIV.PS` (elas são carregadas e armazenadas usando carregamentos e armazenamentos de precisão dupla).

Acenando em direção à importância das aplicações de multimídia, o MIPS64 também inclui instruções multiplicação-adição de inteiro e ponto flutuante: `MADD`, `MADD.S`, `MADD.D` e `MADD.PS`. Os registradores são todos da mesma largura nessas operações combinadas. A Figura A.26 contém uma lista de um subconjunto das operações MIPS64 e seu significado.

Tipo de instrução/opcode	Significado da instrução
Transferências de dados	*Move dados entre registradores e memória ou entre os registradores de inteiro e FP ou especial; apenas o modo de endereço de memória é deslocamento de 16 bits + conteúdo de um GPR*
LB, LBU, SB	Load byte, load byte não sinalizado, store byte (registradores de inteiro to/from)
LH, LHU, SH	Load half word, load half word não sinalizado, store half word (registradores de inteiro to/from)
LW, LWU, SW	Load word, load word não sinalizado, store word (registradores de inteiro to/from)
LD, SD	Load double word, store double word (registradores de inteiro to/from)
L.S, L.D, S.S, S.D	Load SP float, load DP float, store SP float, store DP float
MFC0, MTC0	Copiar de/para GPR para/de um registrador especial
MOV.S, MOV.D	Copiar um registrador FP SP ou DP para outro registrador FP
MFC1, MTC1	Copiar 32 bits para/de registradores FP de/para registradores de inteiro
Aritmética/lógica	*Operações em inteiros ou dados lógicos em GPRs; interceptação de aritmética com sinal no overflow*
DADD, DADDI, DADDU, DADDIU	Add, add immediate (todos os imediatos são de 16 bits); sinalizado e não sinalizado
DSUB, DSUBU	Subtração; sinalizado e não sinalizado
DMUL, DMULU, DDIV, DDIVU, MADD	Multiplicação e divisão, sinalizado e não sinalizado; multiplicação-adição; todas as operações recebem e produzem valores de 64 bits
AND, ANDI	And, and immediate
OR, ORI, XOR, XORI	Or, or immediate, exclusive or, exclusive or immediate
LUI	Load upper immediate; carrega os bits 32 a 47 do registrador com immediate, depois estende o sinal
DSLL, DSRL, DSRA, DSLLV, DSRLV, DSRAV	Shifts: forma immediate (DS__) e variável (DS__V); shifts são shift left logical, right logical, right arithmetic
SLT, SLTI, SLTU, SLTIU	Set less than, set less than immediate; sinalizado e não sinalizado
Controle	*Saltos e desvios condicionais; relativo ao PC ou através do registrador*
BEQZ, BNEZ	GPRs de desvio igual/não igual a zero; offset de 16 bits de PC + 4
BEQ, BNE	GPR de desvio igual/não igual; offset de 16 bits de PC + 4
BC1T, BC1F	Bit de comparação de teste no registrador de estado FP e desvio; offset de 16 bits de PC + 4
MOVN, MOVZ	Copia GPR para outro GPR se o terceiro GPR for negativo, zero
J, JR	Saltos: offset de 26 bits de PC + 4 (J) ou destino no registrador (JR)
JAL, JALR	Salto e link: salva PC + 4 em R31, destino é relativo ao PC (JAL) ou um registrador (JALR)
TRAP	Transfere para o sistema operacional em um endereço vetorizado
ERET	Retorna para o código do usuário de uma exceção; recupera o modo de usuário
Ponto flutuante	*Operações de FP nos formatos DP e SP*
ADD.D, ADD.S, ADD.PS	Soma números DP, SP e pares de números SP
SUB.D, SUB.S, SUB.PS	Subtrai números DP, SP e pares de números SP
MUL.D, MUL.S, MUL.PS	Multiplica ponto flutuante DP, SP e pares de números SP
MADD.D, MADD.S, MADD.PS	Multiplica-soma números DP, SP e pares de números SP
DIV.D, DIV.S, DIV.PS	Divide ponto flutuante DP, SP e pares de números SP
CVT._._	Converte instruções: CVT.x.y converte do tipo x para o tipo y, onde x e y são L (inteiro de 64 bits), W (inteiro de 32 bits), D (DP) ou S (SP). Ambos os operandos são FPRs.
C.__.D,C.__.S	Comparações DP e SP: "__" = LT,GT,LE,GE,EQ,NE; define o bit no registrador de estado FP

FIGURA A.26 Subconjunto das instruções no MIPS64.

A Figura A.22 lista os formatos dessas instruções. SP = precisão única; DP = precisão dupla.

Uso do conjunto de instruções MIPS

Para dar uma ideia de quais instruções são comuns, a Figura A.27 mostra a frequência das instruções e classes de instrução para cinco programas SPECint2000, e a Figura A.28 mostra os mesmos dados para cinco programas SPECfp2000.

A.10 FALÁCIAS E ARMADILHAS

Os arquitetos repetidamente tropeçam em crenças comuns, mas errôneas. Nesta seção veremos algumas delas.

Instrução	gap	gcc	gzip	mcf	perlbmk	Média de inteiro
load	26,5%	25,1%	20,1%	30,3%	28,7%	26%
store	10,3%	13,2%	5,1%	4,3%	16,2%	10%
add	21,1%	19,0%	26,9%	10,1%	16,7%	19%
sub	1,7%	2,2%	5,1%	3,7%	2,5%	3%
mul	1,4%	0,1%				0%
compare	2,8%	6,1%	6,6%	6,3%	3,8%	5%
load imm	4,8%	2,5%	1,5%	0,1%	1,7%	2%
cond desvio	9,3%	12,1%	11,0%	17,5%	10,9%	12%
cond move	0,4%	0,6%	1,1%	0,1%	1,9%	1%
salto	0,8%	0,7%	0,8%	0,7%	1,7%	1%
call	1,6%	0,6%	0,4%	3,2%	1,1%	1%
return	1,6%	0,6%	0,4%	3,2%	1,1%	1%
shift	3,8%	1,1%	2,1%	1,1%	0,5%	2%
AND	4,3%	4,6%	9,4%	0,2%	1,2%	4%
OR	7,9%	8,5%	4,8%	17,6%	8,7%	9%
XOR	1,8%	2,1%	4,4%	1,5%	2,8%	3%
other logical	0,1%	0,4%	0,1%	0,1%	0,3%	0%
load FP						0%
store FP						0%
add FP						0%
sub FP						0%
mul FP						0%
div FP						0%
mov reg-reg FP						0%
compare FP						0%
cond mov FP						0%
other FP						0%

FIGURA A.27 Mix de instrução dinâmica MIPS para cinco programas SPECint2000.
Observe que as instruções move registrador-registrador de inteiro são incluídas na instrução OR. Entradas em branco possuem o valor 0,0%.

Instrução	applu	art	equake	lucas	swim	Média de FP
load	13,8%	18,1%	22,3%	10,6%	9,1%	15%
store	2,9%		0,8%	3,4%	1,3%	2%
add	30,4%	30,1%	17,4%	11,1%	24,4%	23%
sub	2,5%		0,1%	2,1%	3,8%	2%
mul	2,3%			1,2%		1%
compare		7,4%	2,1%			2%
load imm	13,7%		1,0%	1,8%	9,4%	5%
cond desvio	2,5%	11,5%	2,9%	0,6%	1,3%	4%
cond move		0,3%	0,1%			0%
salto			0,1%			0%
call			0,7%			0%
return			0,7%			0%
shift	0,7%		0,2%	1,9%		1%
AND			0,2%	1,8%		0%
OR	0,8%	1,1%	2,3%	1,0%	7,2%	2%
XOR		3,2%	0,1%			1%
other logical			0,1%			0%
carregamento FP	11,4%	12,0%	19,7%	16,2%	16,8%	15%
armazenamento FP	4,2%	4,5%	2,7%	18,2%	5,0%	7%
add FP	2,3%	4,5%	9,8%	8,2%	9,0%	7%
sub FP	2,9%		1,3%	7,6%	4,7%	3%
mul FP	8,6%	4,1%	12,9%	9,4%	6,9%	8%
div FP	0,3%	0,6%	0,5%		0,3%	0%
mov reg-reg FP	0,7%	0,9%	1,2%	1,8%	0,9%	1%
compare FP		0,9%	0,6%	0,8%		0%
cond mov FP		0,6%		0,8%		0%
other FP				1,6%		0%

FIGURA A.28 **Mix de instrução dinâmica MIPS para cinco programas do SPECfp2000.**
Observe que as instruções move registrador-registrador de inteiro são incluídas na instrução OR. Entradas em branco possuem o valor 0,0%.

Armadilha. *Projetar um recurso de conjunto de instruções de "alto nível" especificamente orientado para dar suporte a uma estrutura de linguagem de alto nível.*

As tentativas de incorporar recursos de linguagem de alto nível no conjunto de instruções levaram os arquitetos a disponibilizar instruções poderosas com grande flexibilidade. Entretanto, muitas vezes, essas instruções fazem mais do que é necessário no caso frequente ou não atendem exatamente às necessidades de algumas linguagens. Muitos desses esforços visavam à eliminação do que, na década de 1970, era chamado *semantic gap*. Embora a

ideia seja suplementar o conjunto de instruções com adições que trazem o hardware até o nível da linguagem, as adições podem gerar o que Wulf (1981) chamou de *semantic clash*:

> [...] fornecendo conteúdo semântico excessivo à instrução, o projetista de computador possibilitava o uso da instrução apenas em contextos limitados [pág. 43].

Muitas vezes, as instruções são simplesmente arrasadoras — elas são gerais demais para o caso mais frequente, resultando em trabalho desnecessário e uma instrução mais lenta. Novamente, a CALLS do VAX é um bom exemplo. A CALLS usa uma estratégia de gravação pelo procedimento chamado (os registradores a serem salvos são especificados pelo procedimento chamado), *mas* o salvamento é feito pela instrução call no de quem chama. A instrução CALLS começa com os argumentos colocados na pilha e, depois, realiza as seguintes etapas:

1. Alinhar a pilha, se necessário.
2. Colocar a contagem de argumento na pilha.
3. Salvar os registradores indicados pela máscara de chamada de procedimento na pilha (como mencionado na Seção A.8). A máscara é mantida no código do procedimento chamado — isso permite que o procedimento chamado especifique os registradores a serem salvos por quem chamou, mesmo com uma compilação separada.
4. Colocar o endereço de retorno na pilha e, depois, enviar o topo e a base dos ponteiros da pilha (para o registro de ativação).
5. Limpar os códigos de condição, o que define a habilitação de trap para um status conhecido.
6. Colocar uma palavra para informação de status e uma palavra zero na pilha.
7. Atualizar os dois ponteiros de pilha.
8. Desviar para a primeira instrução do procedimento.

A grande maioria das chamadas nos programas reais não exige essa quantidade de overhead. A maioria dos procedimentos conhece suas contagens de argumento, e uma convenção de encadeamento muito mais rápida pode ser estabelecida usando registradores para passar argumentos em vez da pilha na memória. Além disso, a instrução CALLS força dois registradores a serem usados para o encadeamento, enquanto muitas linguagens exigem apenas um registrador de encadeamento. Muitas tentativas de aceitar gerenciamento de chamada de procedimento e de ativação de pilha falharam, quer porque não atendiam às necessidades da linguagem, quer porque eram muito gerais e, portanto, muito onerosas para usar.

Os projetistas do VAX forneceram uma instrução simples, JSB, que é muito mais rápida, já que coloca o PC de retorno na pilha e salta para o procedimento. Entretanto, a maioria dos compiladores VAX usa as instruções CALLS mais onerosas. As instruções de chamada foram incluídas na arquitetura para padronizar a convenção de encadeamento de procedimento. Outros computadores padronizaram sua convenção de chamada por acordo entre os projetistas de compilador e sem exigir o overhead de uma instrução de chamada de procedimento complexa e muito geral.

Falácia. *Existe um programa típico.*

Muitas pessoas gostam de acreditar que existe um único programa "típico" que poderia ser usado para projetar um conjunto de instruções ótimo. Por exemplo, veja os benchmarks sintéticos discutidos no Capítulo 1. Os dados neste Apêndice mostram claramente que os programas podem variar de modo significativo na maneira como usam um conjunto de instruções. Por exemplo, a Figura A.29 mostra o mix dos tamanhos da transferência de dados para quatro programas do SPEC2000: seria difícil dizer o que é típico nesses programas. As variações são ainda maiores em um conjunto de instruções que aceita uma classe de aplicações, como instruções de decimal, que não são usadas por outras aplicações.

Armadilha. *Inovar na arquitetura de conjunto de instruções para reduzir o tamanho do código sem levar em conta o compilador.*

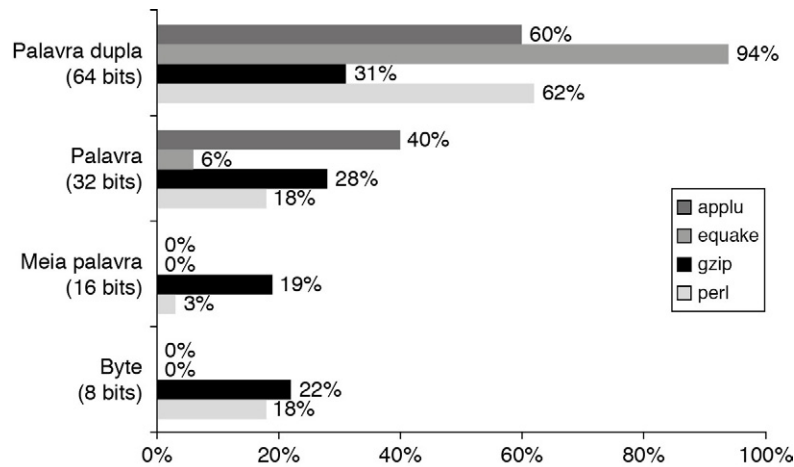



FIGURA A.29 Tamanho da referência de dados de quatro programas do SPEC2000.
Embora seja possível calcular um tamanho médio, seria difícil afirmar que a média é típica dos programas.

A Figura A.30 mostra os tamanhos de código relativos de quatro compiladores para o conjunto de instruções MIPS. Enquanto os arquitetos se esforçam para reduzir o tamanho de código em 30-40%, diferentes estratégias de compilador podem mudar o tamanho de código por fatores muito maiores. Assim como no que se refere às técnicas de otimização de desempenho, o arquiteto deve começar com o código mais compacto que os compiladores podem produzir antes de propor inovações de hardware para ganhar espaço.

Falácia. *Uma arquitetura com falhas não pode ser bem-sucedida.*

O 80x86 oferece um exemplo drástico: a arquitetura de conjunto de instruções é uma que só poderia ser adorada por seus criadores (Apêndice K). Sucessivas gerações de engenheiros da Intel tentaram corrigir as decisões arquitetônicas impopulares tomadas no projeto do 80x86. Por exemplo, a arquitetura aceita a segmentação, enquanto todas as outras escolheram a paginação; ela usa acumuladores estendidos para dados de inteiro, enquanto outros processadores usam registradores de propósito geral; ela usa uma pilha para dados de ponto flutuante, quando todo mundo abandonou as pilhas de execução há muito tempo.

Compilador	Apogee Software Versão 4.1	Green Hills Multi2000 Versão 2.0	Algorithmics SDE4.0B	IDT/c 7.2.1
Arquitetura	MIPS IV	MIPS IV	MIPS 32	MIPS 32
Processador	NEC VR5432	NEC VR5000	IDT 32334	IDT 79RC32364
Kernel de autocorrelação	1,0	2,1	1,1	2,7
Kernel de codificador convolucional	1,0	1,9	1,2	2,4
Kernel de alocação de bit de ponto fixo	1,0	2,0	1,2	2,3
Kernel de FFT complexo de ponto fixo	1,0	1,1	2,7	1,8
Kernel de decodificador Viterbi GSM	1,0	1,7	0,8	1,1
Média geométrica dos cinco kernels	1,0	1,7	1,4	2,0

FIGURA A.30 Tamanho de código relativo para o compilador C Apogee Software Versão 4.1 para aplicação da Telecom dos benchmarks EEMBC.
As arquiteturas de conjunto de instruções são praticamente idênticas, embora os tamanhos de código variem por fatores de 2. Esses resultados foram relatados de fevereiro a junho de 2000.

Em que pesem essas grandes dificuldades, a arquitetura 80x86 foi enormemente bem-sucedida. As razões são três: 1) sua seleção como o microprocessador no IBM PC inicial torna a compatibilidade binária 80x86 extremamente valiosa; 2) a lei de Moore forneceu recursos suficientes para os microprocessadores 80x86 traduzirem para um conjunto de instruções RISC interno e, depois, executarem instruções tipo RISC. Esse mix permite a compatibilidade binária com a valiosa base de software PC e desempenho compatível com processadores RISC; 3) os altíssimos volumes dos microprocessadores de PC significam que a Intel pode facilmente pagar pelo maior custo de projeto da tradução de hardware. Além disso, os altos volumes permitem que o fabricante suba a curva de aprendizagem, o que baixa o custo do produto.

O maior tamanho do die e o aumento do consumo de potência para translação são problemas para as aplicações embarcadas, mas isso faz um tremendo sentido econômico para o desktop. E sua relação custo-desempenho no desktop também o torna atraente para servidores, com sua principal fraqueza para servidores de 32 bits de endereços, o que foi resolvido com os endereços de 64 bits do AMD64 (Cap. 2).

Falácia. *Você pode projetar uma arquitetura sem falhas.*

Todo projeto de arquitetura envolve compromissos feitos no contexto de um conjunto de tecnologias de hardware e software. Com o passar do tempo, essas tecnologias costumam mudar e as decisões que podem ter sido corretas na época em que foram tomadas parecem ter sido equivocadas. Por exemplo, em 1975, os projetistas do VAX enfatizaram demais a importância da eficiência de tamanho de código, subestimando a importância que a facilidade de decodificação e o pipelining teriam cinco anos depois. Um exemplo no campo do RISC é o desvio atrasado (delayed branch) (Apêndice K). Foi uma simples questão de controlar os hazards de pipeline, com pipeline de cinco estágios, mas foi um problema para processadores com pipelines mais longos que terminam a execução de múltiplas instruções por ciclo de clock. Além do mais, quase todas as arquiteturas posteriormente sucumbem à falta de espaço de endereço suficiente.

Em geral, evitar essas falhas em longo prazo provavelmente significaria comprometer a eficiência da arquitetura em curto prazo, o que é arriscado, já que uma nova arquitetura de conjunto de instruções precisa lutar para sobreviver aos seus primeiros anos.

A.11 COMENTÁRIOS FINAIS

As primeiras arquiteturas eram limitadas em seus conjuntos de instruções pela arquitetura de hardware da época. Tão logo a tecnologia de hardware permitia, os arquitetos de computadores começavam a procurar meios de oferecer suporte a linguagens de alto nível. Essa procura levou a três períodos distintos de pensamento sobre como dar um suporte eficiente aos programas. Na década de 1960, as arquiteturas de pilha se tornaram populares. Elas eram vistas como uma boa opção para linguagens de alto nível — e provavelmente eram, dada a tecnologia dos compiladores da época. Na década de 1970, a principal preocupação dos arquitetos era como reduzir custos de software. Essa preocupação foi satisfeita principalmente substituindo software por hardware ou fornecendo arquiteturas de alto nível que poderiam simplificar a tarefa dos projetistas de software. O resultado foi migração para arquitetura de computador de linguagem de alto nível e para arquiteturas poderosas como VAX, que tinha grande número de modos de endereçamento, diversos tipos de dados e uma arquitetura altamente ortogonal. Na década de 1980, uma tecnologia de compilador mais sofisticada e uma nova ênfase no desempenho do processador provocaram um retorno às arquiteturas mais simples, com base principalmente no estilo carregamento-armazenamento de computador.

As seguintes mudanças de arquitetura de conjunto de instruções ocorreram na década de 1990:

- *Duplicações no tamanho de endereço.* Os conjuntos de instruções de endereço de 32 bits para a maioria dos processadores de desktops e servidores foram estendidos para endereços de 64 bits, aumentando o tamanho dos registradores (entre outras coisas) para 64 bits. O Apêndice K fornece três exemplos de arquiteturas que foram de 32 bits para 64 bits.
- *Otimização dos desvios condicionais através de execução condicional.* Nos Capítulos 2 e 3, vimos que os desvios condicionais podem limitar o desempenho dos projetos agressivos de computador. Portanto, houve interesse em substituir os desvios condicionais por conclusão condicional das operações, como movimentação condicional (Apêndice H), o que foi acrescentado na maioria dos conjuntos de instruções.
- *Otimização do desempenho da cache através de pré-busca*(*prefetch*). O Capítulo 2 explica o papel cada vez maior da hierarquia de memória no desempenho dos computadores, com falta de cache em alguns computadores exigindo tantos tempos de instrução quantas falhas de página nos computadores antigos. Portanto, as instruções de pré-busca foram acrescentadas para tentar ocultar o custo das faltas de cache pela realização da busca prévia (Cap. 2).
- *Suporte para multimídia.* A maioria dos conjuntos de instruções de desktops e embarcados foi estendida com suporte para aplicações de multimídia.
- *Operações de ponto flutuante mais rápidas.* O Apêndice J descreve as operações acrescentadas para melhorar o desempenho de ponto flutuante, como operações que realizam uma multiplicação e uma adição com uma única execução (essas operações são incluídas no MIPS).

Entre 1970 e 1985, muitos achavam que o principal trabalho do arquiteto de computador era o projeto de conjuntos de instruções. Como resultado, os livros dessa época enfatizam o projeto de conjunto de instruções, assim como os livros das décadas de 1950 e 1960 enfatizam a aritmética de computador. Esperava-se que o arquiteto inteligente tivesse opiniões sobre os pontos fortes e, especialmente, os pontos fracos dos computadores comuns. A importância da compatibilidade binária em anular inovações no projeto de conjunto de instruções não era reconhecida por muitos pesquisadores e escritores de livros, dando a impressão de que muitos arquitetos teriam a chance de projetar um conjunto de instruções.

A definição de arquitetura de computador foi expandida para incluir o projeto e a avaliação do sistema de computador inteiro — não apenas a definição do conjunto de instruções e não apenas o processador — e, portanto, existem muitos tópicos para o arquiteto estudar. Na verdade, o assunto deste apêndice foi um ponto central do livro em sua primeira edição em 1990, mas agora está incluído em um apêndice principalmente como material de referência!

O Apêndice K pode satisfazer os leitores interessados na arquitetura de conjunto de instruções: ele descreve uma variedade de conjuntos de instruções, que são importantes no mercado de hoje ou historicamente importantes, e compara nove computadores de carregamento-armazenamento conhecidos com o MIPS.

A.12 PERSPECTIVA HISTÓRICA E REFERÊNCIAS

A Seção L.4 (disponível on-line) apresenta um estudo sobre a evolução dos conjuntos de instruções e inclui referências de leitura complementar e exploração de tópicos relacionados.

EXERCÍCIOS POR GREGORY D. PETERSON

A.1 [15] <A.9> Calcule o CPI efetivo para o MIPS usando a Figura A.27. Suponha que tenhamos realizado as seguintes medições do CPI médio para os tipos de instruções:

Instrução	Ciclos de clock
Todas as instruções da ULA	1,0
Carregamentos-armazenamentos	1,4
Desvios condicionais	
Tomados	2,0
Não tomados	1,5
Saltos	1,2

Suponha que 60% dos desvios condicionais sejam tomados e que todas as instruções na categoria "outros" da Figura A.27 sejam instruções da ULA. Tire a média das frequências de instrução de lacuna e gcc para obter o mix de instruções.

A.2 [15] <A.9> Calcule o CPI efetivo para MIPS usando a Figura A.27 e a tabela anterior. Tire a média das frequências de instrução de gzip e perlbmk para obter o mix de instruções.

A.3 [20] <A.9> Calcule o CPI efetivo para o MIPS usando a Figura A.28. Suponha que tenhamos realizado as seguintes medições do CPI médio para os tipos de instruções:

Instrução	Ciclos de clock
Todas as instruções da ULA	1,0
Carregamentos-armazenamentos	1,4
Desvios condicionais:	
Tomados	2,0
Não tomados	1,5
Saltos	1,2
Multiplicação de PF	6,0
Soma de PF	4,0
Divisão de PF	20,0
Carregamento-armazenamento de PF	1,5
Outros PF	2,0

Suponha que 60% dos desvios condicionais sejam tomados e que todas as instruções na categoria "outros" da Figura A.28 sejam instruções da ULA. Tire a média das frequências de instrução de lucas e swim para obter o mix de instruções.

A.4 [20] <A.9> Calcule o CPI efetivo para MIPS usando a Figura A.28 e a tabela anterior. Tire a média das frequências de instrução de applu e art para obter o mix de instruções.

A.5 [10] <A.8> Considere esta sequência de código de alto nível com três declarações:

```
A = B + C;
B = A + C;
D = A - B;
```

Use a técnica de propagação de cópia (Fig. A.20) para transformar a sequência de código até o ponto em que nenhum operando seja um valor calculado. Observe as instâncias nas quais a transformação reduziu o trabalho computacional de uma declaração e os casos em que o trabalho foi aumentado. O que isso sugere sobre o desafio técnico enfrentado quando se tenta satisfazer o desejo de otimizar os compiladores?

A.6 [30] <A.8> As otimizações de compilador podem resultar em melhorias reais no tamanho e/ou desempenho do código. Considere um ou mais dos programas de benchmark da suíte SPEC CPU2006. Use um processador disponível para você e o compilador GNU C para otimizar o programa usando nenhuma otimização, -O1, -O2 e -O3. Compare o desempenho e o tamanho dos programas resultantes. Compare também seus resultados com a Figura A.21.

A.7 [20/20] <A.2, A.9> Considere o seguinte fragmento de código C:

```
para (i = 0; i <= 100; i++)
{ A[i] = B[i] + C; }
```

Suponha que A e B sejam arrays de inteiros de 64 bits, e C e i sejam inteiros de 64 bits. Suponha que todos os valores de dados e seus endereços sejam mantidos na memória (nos endereços 1.000, 3.000, 5.000 e 7.000 para `A`, `B`, `C` e `i`, respectivamente), exceto quando eles estão sendo operados. Suponha que os valores nos registradores sejam perdidos entre iterações do loop.

a. [20] <A.2, A.9> Escreva o código para MIPS. Quantas instruções são necessárias dinamicamente? Quantas referências de dados de memória serão executadas? Qual é o tamanho do código em bytes?

b. [20] <A.2> Escreva o código para x86. Quantas instruções são necessárias dinamicamente? Quantas referências de dados de memória serão executadas? Qual é o tamanho do código em bytes?

A.8 [10/10/10] <A.2, A.7> Para o seguinte, consideramos a codificação de instruções para arquiteturas de conjunto de instruções.

a. [10] <A.2, A.7> Considere o caso de um processador com comprimento de instrução de 12 bits e com 32 registradores de uso geral, de modo que o tamanho dos campos de endereço é 5 bits. É possível ter codificações de instrução para o seguinte?

- 3 instruções com endereço dois
- 30 instruções com endereço um
- 45 instruções com endereço zero

b. [10] <A.2, A.7> Supondo o mesmo comprimento de instrução e tamanhos de campo de dados anteriores, determine se é possível ter:

- 3 instruções com endereço dois
- 31 instruções com endereço um
- 35 instruções com endereço zero

Explique sua resposta.

c. [10] <A.2, A.7> Suponha o mesmo comprimento de instrução e tamanhos de campo de dados acima. Além disso, suponha que já haja três instruções com endereço dois e 24 instruções com endereço zero. Qual é o número máximo de instruções de endereço um que podem ser codificadas para esse processador?

A.9 [10/15] <A.2> Para o seguinte, suponha que os valores `A`, `B`, `C`, `D`, `E` e `F` residam na memória. Suponha também que os códigos de operação de instrução sejam representados em 8 bits, os endereços de memória tenham 64 bits, e os endereços de registrador tenham 6 bits.

a. [10] <A.2> Para cada arquitetura de conjunto de instruções mostrada da Figura A.2, quantos endereços, ou nomes, aparecem em cada instrução para o código calcular `C` = `A` + `B`, e qual é o tamanho total do código?

b. [15] <A.2> Algumas das arquiteturas de conjunto de instrução na Figura A.2 destroem operandos durante o curso do cálculo. Essa perda de valores

de dados no armazenamento interno do processador tem consequências no desempenho. Para cada arquitetura na Figura A.2, escreva a sequência de código para calcular:

```
C = A + B
D = A - E
F = C + D
```

No seu código, marque cada operando destruído durante a execução e marque cada instrução "overhead" que é incluída só para superar essa perda de dados no armazenamento interno do processador. Qual é o tamanho total do código, o número de bytes de instrução e dados movidos para/da memória, o número de instruções overhead e o número de bytes de dados overhead para cada uma das suas sequências de código?

A.10 [20] <A.2, A.7, A.9> O projeto do MIPS fornece 32 registradores de uso geral e 32 registradores de ponto flutuante. Se os registradores são bons, mais registradores são melhores? Liste e discuta o máximo de trade-offs que seriam considerados por projetistas de arquitetura de conjunto de instruções examinando se devem, e o quanto devem, aumentar o número de registradores MIPS.

A.11 [5] <A.3> Considere um struct C que inclua os seguintes membros:

```
struct foo {
       char a;
       bool b;
       int c;
       double d;
       short e;
       float f;
       double g;
       char * cptr;
       float * fptr;
       int x;
};
```

Para uma máquina de 32 bits, qual é o tamanho do struct foo? Qual é o tamanho mínimo necessário para esse struct, supondo que você possa organizar a ordem dos membros struct como quiser? E para uma máquina de 64 bits?

A.12 [30] <A.7> Hoje, muitos fabricantes de computadores incluem ferramentas ou simuladores para permitir a medição do uso do conjunto de instruções de um programa de usuário. Entre os métodos em uso estão a simulação de máquina, o trapping suportado por hardware e uma técnica de compilador que instrumenta o módulo de código de objeto inserindo contadores. Encontre um processador disponível para você que inclua tal ferramenta. Use-o para medir o mix de conjunto de instruções para um dos benchmarks SPEC CPU2006. Compare os resultados aos mostrados neste capítulo.

A.13 [30] <A.8> Processadores mais novos com o Intel i7 Sandy Bridge incluem suporte para instruções de vetor/multimídia AVX. Escreva uma função de multiplicação de matriz densa usando valores de precisão simples e compile-a com diferentes compiladores e flags de otimização. Códigos de álgebra linear usando rotinas Basic Linear Algebra Subroutine (BLAS) como SGEMM incluem versões otimizadas de multiplicação de matriz densa. Compare o tamanho e o desempenho do seu código ao do BLAS SGEMM. Explore o que acontece quando se usam valores de precisão dupla e DGEMM.

A.14 [30] <A.8> Para o código SGEMM desenvolvido anteriormente para o processador i7, inclua o uso de intrínsecos AVX para melhorar o desempenho. Em particular, tente vetorizar seu código para utilizar melhor o hardware AVX. Compare o tamanho e o desempenho do código aos do código original.

A.15 [30] <A.7, A.9> SPIM é um simulador popular para simular processadores MIPS. Use o SPIM para medir o mix de conjunto de instruções para alguns programas do benchmark SPEC CPU2006.

A.16 [35/35/35/35] <A.2-A.8> O gcc visa à maioria das arquiteturas de conjunto de instrução modernas (www.gnu.org/software/gcc/). Crie uma versão do gcc para diversas arquiteturas a que você tenha acesso, como x86, MIPS, PowerPC ARM.

- **a.** [35] <A.2-A.8> Compile um subconjunto de benchmarks inteiros do SPEC CPU2006 e crie uma tabela de tamanhos de código. Que arquitetura é melhor para cada programa?
- **b.** [35] <A.2-A.8> Compile um subconjunto de benchmarks de ponto flutuante do SPEC CPU2006 e crie uma tabela de tamanhos de código. Que arquitetura é melhor para cada programa?
- **c.** [35] <A.2-A.8> Compile um subconjunto de benchmarks EEMBC AutoBench (www.eembc.org/home.php) e crie uma tabela de tamanhos de código. Que arquitetura é melhor para cada programa?
- **d.** [35] <A.2-A.8> Compile um subconjunto de benchmarks de ponto flutuante do EEMBC FPBench e crie uma tabela de tamanhos de código. Que arquitetura é melhor para cada programa?

A.17 [40] <A.2-A.8> A eficiência energética se tornou muito importante para os processadores modernos, especialmente para sistemas embutidos. Crie uma versão do gcc para duas arquiteturas a que você tenha acesso, como x86, MIPS, PowerPC ARM. Compile um subconjunto de benchmarks EEMBC usando o EnergyBench para medir o uso de energia durante a execução. Compare o tamanho de código, o desempenho e o uso de energia para os processadores. Qual é melhor para cada programa?

A.18 [20/15/15/20] Sua tarefa é comparar a eficiência de memória de quatro diferentes estilos de arquitetura de conjunto de instrução. Os estilos de arquitetura são:

- *Acumulador*. Todas as operações acontecem entre um único registrador e um local de memória.
- *Memória-memória*. Todos os endereços de instruções se referem somente a locais de memória.
- *Pilha*. Todas as operações ocorrem no topo da pilha. Push e pop são as únicas instruções que acessam a memória. Todas as outras removem seus operandos da pilha e os substituem com o resultado. A implementação usa uma pilha hardwired somente para as duas primeiras entradas da pilha, o que mantém o circuito do processador muito pequeno e de baixo custo. Posições adicionais de pilha são mantidas em locais de memória, e acessos a essas posições de pilha demandam referências de memória.
- *Carregamento-armazenamento*. Todas as operações ocorrem nos registradores, e instruções registrador para registrador têm três nomes de registrador por instrução.

Para medir a eficiência da memória, faça as seguintes suposições sobre os quatros conjuntos de instruções:

- Todas as instruções têm um número inteiro de bytes de comprimento.
- O opcode é sempre um byte (8 bits).
- Os acessos de memória usam endereçamento direto ou absoluto.
- As variáveis `A`, `B`, `C` e `D` estão inicialmente na memória.

a. [20] <A.2, A.3> Invente seus próprios mnemônicos de linguagem assembly (a Figura A.2 dá uma amostra útil para generalizar), e para cada arquitetura escreva o melhor equivalente em código de linguagem assembly para essa sequência de código de alto nível.

```
A = B + C;
B = A + C;
D = A - B;
```

b. [15] <A.3> Etiquete cada instância dos seus códigos assembly para o item *a* em que um valor seja carregado da memória depois de ter sido carregado uma vez. Etiquete também cada instância do seu código em que o resultado de uma instrução seja enviado para outra instrução como operando e classifique esses eventos como envolvendo armazenamento dentro do processador ou na memória.

c. [15] <A.7> Suponha que a sequência de código dada seja de uma aplicação de computador pequena e embutida, como um controlador de forno de micro-ondas, que usa um endereço de memória de 16 bits e operandos de dados. Se uma arquitetura carregamento-armazenamento for usada, suponha que ela tenha 16 registradores de uso geral. Para cada arquitetura, responda às seguintes perguntas: Quantos bytes de instruções são buscados? Quantos bytes de dados são transferidas da/para a memória? Que arquitetura é a mais eficiente como medida em tráfego total de memória (código + dados)?

d. [20] <A.7> Agora suponha um processador com endereços de memória e operandos de dados de 64 bits. Para cada arquitetura, responda às perguntas do item *c*: como os méritos relativos das arquiteturas mudaram para as métricas selecionadas?

A.19 [30] <A.2, A.3> Use os quatro estilos de arquitetura de conjunto de instruções anterior, mas suponha que as operações de memória incluam registradores indiretos além de endereçamento direto. Invente seus próprios mnemônicos de linguagem assembly (a Figura A.2 dá uma amostra útil para generalizar), e para cada arquitetura escreva o melhor equivalente em código de linguagem assembly para este fragmento de código C:

```
para (i = 0; i <= 100; i++)
{ A[i] = B[i] + C; }
```

Suponha que A e B sejam arrays de inteiros de 64 bits, e C e i sejam inteiros de 64 bits.

A segunda e a terceira colunas contêm a porcentagem cumulativa das referências de dados e desvios, respectivamente, que podem ser acomodadas com o número correspondente de bits de magnitude no deslocamento. Essas são as distâncias médias de todos os programas de inteiros e de ponto flutuante nas Figuras A.8 e A.15.

A.20 [20/20/20] <A.3> Estamos projetando formatos de conjunto de instrução para uma arquitetura carregamento-armazenamento e estamos tentando decidir se vale a pena ter múltiplos comprimentos offset para desvios e referências de memória. O comprimento de uma instrução seria igual a 16 bits + comprimento de offset em bits, então as instruções da ULA seriam de 16 bits. A Figura A.31 contém dados sobre o tamanho de offset para a arquitetura Alpha com otimização total para o SPEC

Número de bits de magnitude de offset	Referências de dados cumulativas	Desvios cumulativos
0	30,4%	0,1%
1	33,5%	2,8%
2	35,0%	10,5%
3	40,0%	22,9%
4	47,3%	36,5%
5	54,5%	57,4%
6	60,4%	72,4%
7	66,9%	85,2%
8	71.6%	90,5%
9	73,3%	93,1%
10	74,2%	95,1%
11	74,9%	96,0%
12	76,6%	96,8%
13	87,9%	97,4%
14	91,9%	98,1%
15	100%	98,5%
16	100%	99,5%
17	100%	99,8%
18	100%	99,9%
19	100%	100%
20	100%	100%
21	100%	100%

FIGURA A.31 Dados sobre o tamanho de offset para a arquitetura Alpha com otimização total para o SPEC CPU2000.

CPU2000. Para as frequências de conjunto de instrução, use os dados para o MIPS da média dos cinco benchmarks para a máquina carregamento-armazenamento na Figura A.27. Suponha que as instruções diversas sejam todas instruções ALU que usam somente registradores.

a. [20] <A.3> Suponha que os offsets permitidos sejam 0, 8, 16 ou 24 bits de comprimento, incluindo o bit de sinal. Qual é o comprimento médio de uma instrução executada?

b. [20] <A.3> Suponha que queiramos uma instrução de comprimento fixo e escolhemos um comprimento de instrução de 24 bits (para tudo, incluindo as instruções da ULA). Para cada offset de mais de 8 bits são necessárias instruções adicionais. Determine o número de bytes de instrução buscados nessa máquina com tamanho de instrução fixo *versus* aquelas com instrução com tamanho em bytes variável como definida no item *a*.

c. [20] <A.3> Agora suponha que usamos um comprimento de offset fixo de 24 bits de modo que não sejam necessárias instruções adicionais. Quantos bytes de instrução seriam necessários? Compare esse resultado com sua resposta ao item *b*.

A.21 [20/20] <A.3, A.6, A.9> O tamanho dos valores de deslocamento necessários para o modo de endereçamento de deslocamento ou para endereçamento relativo ao PC pode ser extraído das aplicações compiladas. Use um disassembler com um ou mais dos benchmarks SPEC CPU2006 compilados para o processador MIPS.

a. [20] <A.3, A.9> Para cada instrução usando endereçamento de deslocamento, registre o valor de deslocamento usado. Crie um histograma para os valores de deslocamento. Compare os resultados aos mostrados neste capítulo na Figura A.8.

b. [20] <A.6, A.9> Para cada desvio usando endereçamento relacionado ao PC, registre o valor de deslocamento usado. Crie um histograma para os valores de deslocamento. Compare os resultados aos mostrados neste capítulo na Figura A.15.

A.22 [15/15/10/10] <A.3> O valor representado pelo número hexadecimal 434F 4D50 5554 4552 deve ser armazenado em uma palavra dupla de 64 bits.

a. [15] <A.3> Usando a organização física da primeira linha na Figura A.5, escreva o valor a ser armazenado usando a ordem de byte Big Endian. A seguir, interprete cada byte com um caractere ASCII e debaixo de cada byte escreva o caractere correspondente, formando a string de caracteres como ela seria armazenada na ordem Big Endian.

b. [15] <A.3> Usando a mesma organização física do item *a*, escreva o valor a ser armazenado usando a ordem de byte Little Endian; embaixo de cada byte escreva o caractere ASCII correspondente.

c. [10] <A.3> Quais são os valores hexadecimais de todas as palavras de 2 bytes desalinhadas que podem ser lidas a partir da palavra dupla de 64 bits quando armazenadas na ordem de byte Big Endian?

d. [10] <A.3> Quais são os valores hexadecimais de todas as palavras de 4 bytes desalinhadas que podem ser lidas a partir da palavra dupla de 64 bits quando armazenadas na ordem de byte Little Endian?

A.23 [Discussão] <A.2-A.12> Considere aplicações típicas para desktop, servidor, nuvem e computação embutida. Qual seria o impacto sobre a arquitetura de conjunto de instruções para máquinas visando cada um desses mercados?

Revisão da hierarquia da memória

Cache: um local seguro para esconder ou armazenar coisas.
Webster's New World Dictionary of the American Language
Second College Edition (1976)

B.1 INTRODUÇÃO

Este apêndice é uma revisão rápida da hierarquia da memória, incluindo os fundamentos da memória cache e da memória virtual, equações de desempenho e otimizações simples. A primeira seção revisa os 36 termos a seguir:

cache	*totalmente associativo*	*write allocate*
memória virtual	*bit de modificação*	*cache unificada*
ciclos de stall da memória	*offset de bloco*	*faltas por instrução*
mapeamento direto	*write-back*	*bloco*
bit de validade	*cache de dados*	*localidade*
endereço de bloco	*tempo de acerto*	*rastreio de endereço*
write-through	*falta na cache*	*conjunto*
cache de instrução	*falta de página*	*substituição aleatória*
tempo médio de acesso à memória	*taxa de falta*	*campo de índice*
acerto na cache	*associativo por conjunto com n vias*	*no-write allocate*
página	*uso menos recente*	*buffer de escrita*
penalidade por falta	*campo de tag*	*stall de escrita*

Se esta revisão prosseguir muito rapidamente, você pode examinar o Capítulo 7 do livro *Organização e Projeto de Computador*, que escrevemos para os leitores com menos experiência.

Cache é o nome dado ao nível mais alto ou ao primeiro nível de hierarquia da memória, encontrada quando o endereço sai do processador. Como o princípio de localidade se aplica a muitos níveis — e tirar proveito da localidade para melhorar o desempenho é algo popular —, o termo cache é aplicado sempre que o buffering é empregado para reutilizar itens que ocorrem comumente. Alguns exemplos incluem *caches de arquivos*, *caches de nomes*, e assim por diante.

Quando o processador encontra um item de dados solicitado na cache, isso é chamado *acerto na cache* (cache hit). Quando o processador não encontra na cache o item de dados de que precisa, ocorre uma *falta na cache* (cache miss). Uma coleção de dados de tamanho fixo contendo a palavra solicitada, chamada *bloco* ou linha, é apanhada da memória principal e colocada na cache. *Localidade temporal* nos diz que provavelmente precisamos dessa palavra novamente no futuro próximo e, por isso, é útil colocá-la na cache, onde pode ser acessada rapidamente. Devido à *localidade espacial*, existe alta probabilidade de que os outros dados no bloco logo serão necessários.

O tempo exigido para a falta na cache depende tanto da latência quanto da largura de banda da memória. A latência determina o tempo para apanhar a primeira palavra do bloco, e a largura de banda determina o tempo para apanhar o restante desse bloco. Uma falta na cache é tratada pelo hardware e faz com que os processadores usando a execução em ordem parem ou atrasem até que os dados estejam disponíveis. Com a execução fora de ordem, uma instrução usando o resultado ainda precisa esperar, mas outras instruções podem prosseguir durante a falta.

De modo semelhante, nem todos os objetos referenciados por um programa precisam residir na memória principal. *Memória virtual* significa que alguns objetos podem residir em disco. O espaço de endereço normalmente é desmembrado em blocos de tamanho fixo, chamados *páginas.* A qualquer momento, cada página reside ou na memória principal ou no disco. Quando o processador referencia um item dentro de uma página que não está presente na cache ou na memória principal, ocorre uma *falta de página*, e a página inteira é movida do disco para a memória principal. Como as faltas de página levam muito tempo, elas são tratadas pelo software, e o processador não fica em stall. O processador normalmente passa para alguma outra tarefa enquanto ocorre o acesso ao disco. De um ponto de vista de alto nível, a confiança na localidade das referências e os relacionamentos relativos em tamanho e custo relativo por bit de cache contra a memória principal são semelhantes aos da memória principal *versus* o disco.

A Figura B.1 mostra o intervalo de tamanhos e os tempos de acesso de cada nível na hierarquia de memória para os computadores variando de desktops até servidores.

Revisão de desempenho da cache

Devido à localidade e à velocidade mais alta das memórias menores, uma hierarquia de memória pode melhorar substancialmente o desempenho. Um método para avaliar o desempenho da cache é expandir nossa equação de tempo de execução do processador do Capítulo 1. Agora, levamos em consideração o número de ciclos durante os quais o processador fica em stall, esperando por um acesso à memória, que chamamos *ciclos de stall da memória*. O desempenho é, então, o produto do tempo de ciclo de clock pela soma dos ciclos de processador e ciclos de stall da memória:

$$\text{Tempo de execução da CPU} = (\text{Ciclos de clock da CPU} + \text{Ciclos de stall da memória}) \times \text{Tempo do ciclo de clock}$$

Nível	1	2	3	4
Nome	Registradores	Cache	Memória principal	Armazenamento em disco
Tamanho típico	< 1 KB	< 32 KB-8 MB	< 512 GB	> 1 TB
Tecnologia de implementação	Memória própria com múltiplas portas, CMOS	CMOS SRAM no chip ou fora	CMOS DRAM	Disco magnético
Tempo de acesso (ns)	0,15-0,30	0,5-15	30-200	5.000.000
Largura de banda (MB/s)	100.000-1.000,000	10.000-40.000	5.000-20.000	50-500
Gerenciado por	Compilador	Hardware	Sistema operacional	Sistema operacional/operador
Com apoio de	Cache	Memória principal	Disco	CD ou fita

FIGURA B.1 Os níveis típicos na hierarquia ficam mais lentos e maiores enquanto nos afastamos do processador para uma estação de trabalho grande ou servidor pequeno.
Os computadores embarcados não devem ter armazenamento em disco e memórias e caches muito menores. Os tempos de acesso aumentam à medida que passamos para níveis mais baixos da hierarquia, tornando viável gerenciar a transferência de modo mais simples. A tecnologia de implementação mostra a tecnologia típica usada para essas funções. O tempo de acesso é dado em nanossegundos para os valores típicos em 2006; esses tempos diminuirão com o tempo. A largura de banda é dada em megabytes por segundo entre os níveis na hierarquia de memória. A largura de banda para armazenamento em disco inclui a mídia e as interfaces em buffer.

Esta equação considera que os ciclos de clock da CPU incluem o tempo para tratar de um acerto na cache e que o processador é stalled durante uma falta na cache. A Seção B.2 reexamina essa suposição simplificada.

O número de ciclos de stall da memória depende do número de falta e do custo por falta, o que é chamado *penalidade por fal*ta:

$$
\begin{aligned}
\text{Ciclos de stall da memória} &= \text{Número de faltas} \times \text{penalidade de falta} \\
&= \text{IC} \times \frac{\text{Faltas}}{\text{Instrução}} \times \text{Penalida de falta} \\
&= \text{IC} \times \frac{\text{Acessos à memória}}{\text{Instrução}} \times \text{Taxa de falta} \times \text{Penalidade de falta}
\end{aligned}
$$

A vantagem do último formato é que os componentes podem ser facilmente medidos. Já sabemos como medir o número de instruções (IC — instruction count). (Para processadores especulativos, só contamos instruções que são confirmadas.) A medição do número de referências de memória por instrução pode ser feita no mesmo padrão; cada instrução exige um acesso à instrução, e é fácil decidir se também exige um acesso a dados.

Observe que calculamos a penalidade por falta como uma média, mas a usaremos a seguir como se ela fosse uma constante. A memória por trás da cache pode estar ocupada no momento da falta, devido às solicitações de memória anteriores ou atualização da memória. O número de ciclos de clock também varia nas interfaces entre os diferentes clocks do processador, barramento e memória. Assim, lembre-se de que o uso de um único número para a penalidade por falta é uma simplificação.

O componente *taxa de falta* é simplesmente a fração dos acessos à cache que resultam em uma falta (ou seja, o número de acessos que se perdem dividido pelo número de acessos). As taxas de falta podem ser medidas com simuladores de cache que assumem um *rastreio de endereço* da instrução e referências a dados, simulam o comportamento da cache para determinar quais referências acertam e quais perdem, e depois informam os totais de acerto e falta. Hoje, muitos microprocessadores oferecem hardware para contar o número de faltas e referências de memória, que é um modo muito mais fácil e mais rápido de medir a taxa de falta.

A fórmula anterior é uma aproximação, pois as taxas de falta e as penalidades por falta normalmente são diferentes para leituras e escritas. Os ciclos de clock de stall da memória poderiam, então, ser definidos em termos do número de acessos à memória por instrução, penalidade por falta (em ciclos de clock) para leituras e escritas, e taxa de falta para leituras e escritas:

$$\begin{aligned}\text{Ciclos de clock de stall da memória} = {} & \text{IC} \times \text{Leituras por instrução} \times \text{Taxa de falta na leitura} \\ & \times \text{Penalidade por falta na leitura} + \text{IC} \times \text{Escritas por instrução} \\ & \times \text{Taxa de falta na escrita} \times \text{Penalidade por falta na escrita}\end{aligned}$$

Normalmente, simplificamos a fórmula completa, combinando as leituras e escritas e encontrando as taxas médias de falta e a penalidade por falta para leituras *e* escritas:

$$\text{Ciclos de stall da memória} = \text{IC} \times \frac{\text{Acessos à memória}}{\text{Instrução}} \times \text{Taxa de falta} \times \text{Penalidade de falta}$$

A taxa de falta é uma das medidas mais importantes do projeto de cache, mas, conforme veremos nas próximas seções, não é a única.

Exemplo Considere que temos um computador onde os ciclos de clock por instrução (CPI) seja de 1,0 quando todos os acessos à memória têm acertos na cache. Os únicos acessos a dados são carregamentos e armazenamentos, e estes totalizam 50% das instruções. Se a penalidade por falta for 25 ciclos de clock e a taxa de falta for 2%, quão mais rápido o computador será se todas as instruções forem acertos de cache?

Resposta Primeiro, calcule o desempenho para o computador que sempre acerta:

$$\begin{aligned}\text{Tempo de execução da CPU} &= (\text{Ciclos de clock da CPU} + \text{Ciclos de stall da memória}) \\ &\quad \times \text{Ciclo de clock} \\ &= (\text{IC} \times \text{CPI} + 0) \times \text{Ciclo de clock} \\ &= (\text{IC} \times 1{,}0 \times \text{Ciclo de clock})\end{aligned}$$

Agora, para o computador com a cache real, primeiro calculamos os ciclos de stall da memória:

$$\begin{aligned}\text{Ciclos de stall da memória} &= \text{IC} \times \frac{\text{Acesso à memória}}{\text{Instrução}} \times \text{Taxa de falta} \times \text{Penalidade de falta} \\ &= \text{IC} \times (1 + 0{,}5) \times 0{,}02 \times 25 \\ &= \text{IC} \times 0{,}75\end{aligned}$$

onde o termo do meio (1 + 0,5) representa um acesso à instrução e 0,5 acesso a dados por instrução. O desempenho total é, então:

$$\begin{aligned}\text{Tempo execução CPU}_{\text{cache}} &= (\text{IC} \times 1{,}0 + \text{IC} \times 0{,}75) \times \text{Ciclo de clock} \\ &= 1{,}75 \times \text{IC} \times \text{Ciclo de clock}\end{aligned}$$

A razão de desempenho é o inverso dos tempos de execução:

$$\frac{\text{Tempo execução CPU}_{\text{cache}}}{\text{Tempo execução CPU}} = \frac{1{,}75 \times \text{IC} \times \text{Ciclo de clock}}{1{,}0 \times \text{IC} \times \text{Ciclo de clock}} = 1{,}75$$

O computador sem faltas de cache é 1,75 vez mais rápido.

Alguns projetistas preferem medir a taxa de falta como *faltas por instrução* em vez de faltas por referência de memória. Esses dois estão relacionados:

$$\frac{\text{Faltas}}{\text{Instrução}} = \frac{\text{Taxas de falta} \times \text{Acessos à memória}}{\text{Contador de instruções}} = \text{Taxas de falta} \times \frac{\text{Acessos à memória}}{\text{Instrução}}$$

Esta última fórmula é útil quando se conhece o número médio de acessos à memória por instrução, pois permite converter a taxa de falta em faltas por instrução e vice-versa. Por exemplo, podemos transformar a taxa de falta por referência à memória, do exemplo anterior, em faltas por instrução:

$$\frac{\text{Faltas}}{\text{Instrução}} = \text{Taxas de falta} \times \frac{\text{Acesso à memória}}{\text{Instrução}} = 0,02 \times (1,5) = 0,030$$

A propósito, as faltas por instrução normalmente são relatadas como faltas por 1.000 instruções para mostrar inteiros no lugar de frações. Assim, a resposta anterior também poderia ser expressa como 30 faltas por 1.000 instruções.

A vantagem das faltas por instrução é que isso independe da implementação do hardware. Por exemplo, processadores especulativos apanham cerca do dobro das instruções que são realmente confirmadas, o que pode reduzir artificialmente a taxa de falta se for medida como faltas por referência à memória em vez de faltas por instrução. A desvantagem é que as faltas por instrução dependem da arquitetura; por exemplo, o número médio de acessos à memória por instrução pode ser muito diferente para um 80x86 *versus* MIPS. Assim, as faltas por instrução são mais populares com arquitetos que trabalham com uma única família de computadores, embora a semelhança das arquiteturas RISC permita que alguém ofereça ideias para outras.

Exemplo Para mostrar a equivalência entre as duas equações de taxa de falta, vamos refazer o exemplo anterior, dessa vez considerando uma taxa de falta por 1.000 instruções igual a 30. Qual é o tempo de stall da memória em termos de contagem de instruções?

Resposta Recalculando os ciclos de stall da memória:

$$\begin{aligned}
\text{Ciclos de stall da memória} &= \text{Número de faltas} \times \text{Penalidade de falta} \\
&= \text{IC} \times \frac{\text{Faltas}}{\text{Instrução}} \times \text{Penalidade de falta} \\
&= \text{IC}/1.000 \times \frac{\text{Faltas}}{\text{Instrução} \times 1.000} \times \text{Penalidade de falta} \\
&= \text{IC}/1.000 \times 30 \times 25 \\
&= \text{IC}/1.000 \times 750 \\
&= \text{IC} \times 0,75
\end{aligned}$$

Obtemos a mesma resposta da página B-3, mostrando a equivalência das duas equações.

Quatro perguntas sobre hierarquia de memória

Continuamos nossa introdução às caches respondendo às quatro perguntas mais comuns para o primeiro nível da hierarquia de memória:

P1: Onde um bloco pode ser colocado no nível superior? (*posicionamento de bloco*)
P2: Como um bloco é localizado se estiver no nível superior? (*identificação de bloco*)
P3: Qual bloco deverá ser substituído em caso de falta? (*substituição de bloco*)
P4: O que acontece em uma escrita? (*estratégia de escrita*)

As respostas para essas perguntas nos ajudam a entender as diferentes escolhas em relação à memória em diferentes níveis de uma hierarquia; logo, fazemos essas quatro perguntas em cada exemplo.

P1: Onde um bloco pode ser colocado em uma cache?

A Figura B.2 mostra que as restrições sobre o local onde um bloco é colocado criam três categorias de organização de cache:

- Se cada bloco tiver apenas um local onde pode aparecer na cache, essa cache é considerada de *mapeamento direto*. O mapeamento normalmente é

(*Endereço de bloco*) MOD (*Número de blocos na cache*)

- Se um bloco puder ser colocado em qualquer lugar na cache, a cache é considerada *totalmente associativa*.
- Se um bloco puder ser colocado em um conjunto restrito de locais na cache, a cache é *associativa por conjunto*. Um *conjunto* é um grupo de blocos na cache. Um bloco é primeiro mapeado em um conjunto e depois pode ser colocado em qualquer lugar dentro desse conjunto. O conjunto normalmente é escolhido pela *seleção de bits*, ou seja,

(*Endereço de bloco*) MOD (*Número de conjuntos na cache*)

Se houver *n* blocos em um conjunto, o local da cache é chamado *associativo por conjunto com n vias*.

O intervalo de caches de mapeamento direto para totalmente associativo é, na realidade, uma continuidade de níveis de associatividade em conjunto. O mapeamento direto é

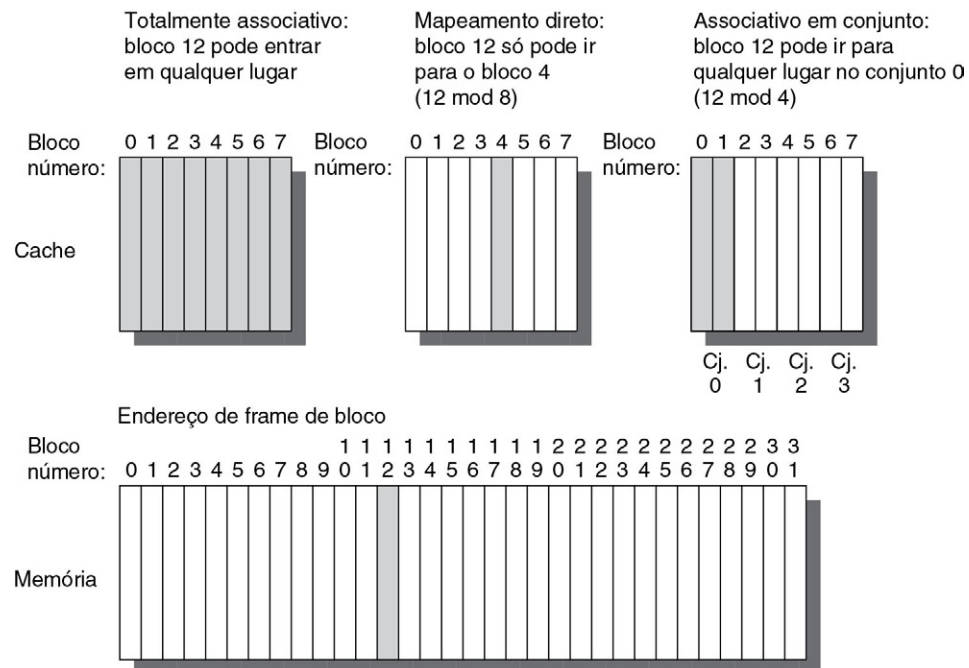



FIGURA B.2 **Essa cache de exemplo possui oito frames em bloco e a memória tem 32 blocos.** As três opções para caches aparecem da esquerda para a direita. Na cache totalmente associativa, o bloco 12 do nível inferior pode entrar em qualquer um dos oito frames de bloco da cache. Com o mapeamento direto, o bloco 12 só pode ser colocado no frame de bloco 4 (12 módulo 8). A cache associativa por conjunto, que possui algo dos dois recursos, permite que o bloco seja colocado em qualquer lugar no bloco 0 ou no bloco 1 da cache. As caches reais contêm milhares de frames de bloco, e as memórias reais contêm milhões de blocos. A organização associativa por conjunto possui quatro conjuntos com dois blocos por conjunto, chamada associativa por conjunto com duas vias. Suponha que não exista nada na cache e que o endereço de bloco em questão identifique o bloco de nível inferior 12.

simplesmente associativo em conjunto com uma via, e uma cache totalmente associativa com *m* blocos poderia ser chamada "associativa por conjunto com *m* vias". De modo equivalente, o mapeamento direto pode ser considerado como tendo *m* conjuntos, e o totalmente associativo como tendo um conjunto.

Hoje, a grande maioria das caches de processadores é mapeada diretamente, associativa por conjunto com duas vias ou associativa por conjunto com quatro vias, por motivos que veremos em breve.

P2: Como um bloco é localizado se estiver na cache?

As caches possuem uma tag de endereço em cada frame de bloco, que indica o endereço do bloco. A tag de cada bloco da cache que poderia conter a informação desejada é verificada, para saber se corresponde ao endereço de bloco do processador. Em geral, todas as tags possíveis são pesquisadas em paralelo, pois a velocidade é crítica.

É preciso haver um meio de saber que um bloco de cache não tem informação válida. O procedimento mais comum é acrescentar um *bit de validade* à tag para dizer se essa entrada contém ou não um endereço válido. Se o bit não for definido, não poderá haver uma combinação nesse endereço.

Antes de passar à próxima pergunta, vamos explorar o relacionamento de um endereço do processador com a cache. A Figura B.3 mostra como um endereço é dividido. A primeira divisão ocorre entre o *endereço de bloco* e o *offset de bloco*. O endereço do frame de bloco pode ser dividido ainda mais, em *campo de tag* e *campo de índice*. O campo de offset de bloco seleciona os dados desejados do bloco, o campo de índice seleciona o conjunto e o campo de tag é comparado com ele para ver se há um acerto. Embora a comparação pudesse ser feita sobre mais de um endereço do que a tag, isso não é necessário, devido ao seguinte:

- O offset não deve ser usado na comparação, pois ou o bloco inteiro está presente ou não, e por isso todos os offsets de bloco resultam em uma combinação por definição.
- A verificação do índice é redundante, pois foi usada para selecionar o conjunto a ser verificado. Um endereço armazenado no conjunto 0, por exemplo, precisa ter 0 no campo de índice ou não poderia ser armazenado no conjunto 0; o conjunto 1 precisa ter um valor de índice igual a 1, e assim por diante. Essa otimização economiza hardware e energia, reduzindo o tamanho da memória para a tag de cache.

Se o tamanho total da cache for mantido igual, aumentar a associatividade aumentará o número de blocos por conjunto, diminuindo assim o tamanho do índice e aumentando o tamanho da tag. Ou seja, o limite de índice de tag na Figura B.3 se move para a direita aumentando-se a associatividade, com a extremidade das caches totalmente associativas não possuindo campo de índice.

Endereço do bloco		Offset de bloco
Tag	Índice	

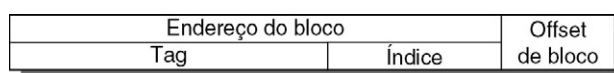



FIGURA B.3 **As três partes de um endereço em uma cache associativa em conjunto ou com mapeamento direto.**
A tag é usada para verificar todos os blocos no conjunto, e o índice é usado para selecionar o conjunto. O offset de bloco é o endereço dos dados desejados dentro do bloco. As caches totalmente associativas não possuem campo de índice.

P3: Qual bloco deverá ser substituído em uma falta na cache?

Quando ocorre uma falta, a controladora de cache precisa selecionar um bloco a ser substituído pelos dados desejados. Um benefício do posicionamento mapeado diretamente é que as decisões de hardware são simplificadas — na verdade, isso é tão simples que não existe escolha: somente um frame de bloco é verificado em busca de um acerto, e somente esse bloco pode ser substituído. Com o posicionamento totalmente associativo ou associativo por conjunto, existem muitos blocos para escolher em uma falta. Existem três estratégias principais empregadas para selecionar qual bloco será substituído:

- *Aleatória.* Para espalhar a alocação uniformemente, os blocos candidatos são selecionados aleatoriamente. Alguns sistemas geram números de bloco pseudoaleatórios para obter um comportamento reprodutível, o que é particularmente útil quando se realiza a depuração do hardware.
- *Uso menos recente* (LRU). Para reduzir a chance de perder informações que serão necessárias em breve, os acessos aos blocos são registrados. Contando com o passado para prever o futuro, o bloco substituído é aquele que não foi utilizado por mais tempo. O LRU conta com um corolário da localidade: se os blocos recém-utilizados provavelmente serão usados de novo, então um bom candidato ao descarte é o bloco usado menos recentemente.
- *Primeiro a entrar, primeiro a sair* (FIFO). Como a estratégia LRU pode ser complicada de calcular, isso é próximo do LRU, determinando o bloco *mais antigo* em vez do LRU.

Uma virtude da substituição aleatória é que ela é simples de criar no hardware. À medida que o número de blocos a registrar aumenta, o LRU torna-se cada vez mais dispendioso e, geralmente, é apenas aproximado. Uma aproximação comum (muitas vezes chamada pseudo-LRU) tem um conjunto de bits para cada conjunto na cache com cada bit correspondendo a uma única via (uma via é um banco em uma cache associativa por conjunto; existem quatro vias em uma cache associativa por conjunto com quatro vias) na cache. Quando um conjunto é acessado, o bit correspondente à via contendo o bloco desejado é ligado. Se todos os bits associados com um conjunto estiverem ligados, eles são resetados, com exceção do bit ligado mais recentemente. Quando um bloco precisa ser substituído, o processador seleciona um bloco da via cujo bit está desligado, muitas vezes aleatoriamente, se mais de uma opção estiver disponível. Isso aproxima o LRU, uma vez que o bloco que é substituído não terá sido acessado desde a última vez que todos os blocos no conjunto foram acessados. A Figura B.4 mostra a diferença em taxas de falta entre a substituição LRU, aleatória e FIFO.

	Associatividade								
	Duas vias			Quatro vias			Oito vias		
Tamanho	LRU	Aleatório	FIFO	LRU	Aleatório	FIFO	LRU	Aleatório	FIFO
16 KB	114,1	117,3	115,5	111,7	115,1	113,3	109,0	111,8	110,4
64 KB	103,4	104,3	103,9	102,4	102,3	103,1	99,7	100,5	100,3
256 KB	92,2	92,1	92,5	92,1	92,1	92,5	92,1	92,1	92,5

FIGURA B.4 Faltas de cache de dados por 1.000 instruções comparando a substituição usada menos recentemente, aleatória e primeiro a entrar, primeiro a sair para vários tamanhos e associatividades.

Há pouca diferença entre LRU e aleatório para a cache de tamanho grande, com LRU sendo superior às outras estratégias em caches menores. O FIFO geralmente é superior à técnica aleatória nos tamanhos de cache menores. Esses dados foram coletados para um tamanho de bloco de 64 bytes, para a arquitetura Alpha, usando 10 benchmarks SPEC2000. Cinco são do SPECint2000 (gap, gcc, gzip, mcf e perl) e cinco são do SPECfp2000 (applu, art, equake, lucas e swim). Usaremos esse computador e esses benchmarks na maioria das figuras deste apêndice.

P4: O que acontece em uma escrita?

As leituras dominam os acessos à cache do processador. Todos os acessos à instrução são leituras, e a maioria das instruções não escreve na memória. As Figuras A.32 e A33 no Apêndice A sugerem uma mistura de 10% de armazenamentos e 26% de carregamentos para os programas MIPS, tornando as escritas 10%/(100% + 26% + 10%) ou cerca de 7% do tráfego geral da memória. Do tráfego da cache *de dados*, as escritas representam 10%/(26% + 10%) ou cerca de 28%. Tornar o caso comum rápido significa otimizar as caches para leituras, especialmente porque os processadores tradicionalmente esperam que as leituras terminem, mas não precisam esperar pelas escritas. Contudo, a lei de Amdahl (Seção 1.9) nos lembra que os projetos de alto desempenho não podem desconsiderar a velocidade das escritas.

Felizmente, o caso comum também é o caso fácil de tornar rápido. O bloco pode ser lido da cache ao mesmo tempo que a tag é lida e comparada, de modo que a leitura do bloco começa assim que o endereço do bloco está disponível. Se a leitura for um acerto, a parte solicitada do bloco é passada para o processador imediatamente. Se for uma falta, não existe um benefício, mas também nenhum prejuízo, exceto mais potência nos computadores desktop e servidor; basta ignorar o valor lido.

Tal otimismo não é permitido para as escritas. A modificação de um bloco não pode ser iniciada até que a tag seja verificada para saber se o endereço é um acerto. Como a verificação de tag não pode ocorrer em paralelo, as escritas normalmente levam mais tempo do que as leituras. Outra complexidade é que o processador também especifica o tamanho da escrita, normalmente entre 1-8 bytes; somente essa parte de um bloco pode ser alterada. Ao contrário, as leituras podem acessar mais bytes do que o necessário sem medo.

As políticas de escrita normalmente distinguem os projetos de cache. Existem duas opções básicas quando se escreve na cache:

- *Write-through.* A informação é escrita tanto no bloco da cache *quanto* no bloco na memória de nível inferior.
- *Write-back.* A informação é escrita somente no bloco da cache. O bloco de cache modificado é escrito na memória principal somente quando for substituído.

Para reduzir a frequência de escrita dos blocos na substituição, normalmente é usado um recurso chamado *bit de modificação* (*dirty bit*). Esse bit de *estado* indica se o bloco está modificado enquanto na cache (*dirty — sujo*) ou não modificado (*clean — limpo*). Se não estiver modificado, o bloco não é escrito de volta em uma falta, pois informações idênticas na cache são encontradas nos níveis inferiores.

Tanto o write-back quanto o write-through possuem vantagens. Com o write-back, as escritas ocorrem na velocidade da memória cache, e várias escritas dentro de um bloco exigem apenas uma escrita na memória de nível inferior. Como algumas escritas não vão para a memória, o write-back usa menos largura de banda de memória, tornando-se atraente nos multiprocessadores. Como o write-back utiliza o restante da hierarquia da memória e menos interconexão de memória do que o write-through, ele também economiza energia, tornando-se atraente para aplicações embarcadas.

O write-through é mais fácil de implementar do que o write-back. A cache sempre está limpa, de modo que, ao contrário do write-back, as faltas de leitura nunca resultam em escritas no nível inferior. O write-through também tem a vantagem de que o próximo nível inferior tem a cópia mais atualizada dos dados, o que simplifica a coerência dos dados. A coerência dos dados é importante para multiprocessadores e para a E/S, que examinamos no Capítulo 4 e no Apêndice D. As caches multiníveis tornam o write-through mais viável para as caches de nível superior, pois as escritas só precisam se propagar para o próximo nível inferior, e não até a memória principal.

Conforme veremos, a E/S e os multiprocessadores são inconstantes: eles querem write-back para caches de processador, para reduzir o tráfego da memória, e write-through para manter a cache coerente com os níveis inferiores da hierarquia de memória.

Quando o processador precisa esperar que as escritas terminem durante o write-through, o processador é considerado *stall de escrita*. Uma otimização comum para reduzir os stalls de escrita é um *buffer de escrita*, que permite ao processador continuar assim que os dados forem escritos no buffer, sobrepondo a execução do processador à atualização da memória. Conforme veremos rapidamente, stalls de escrita podem ocorrer mesmo com buffers de escrita.

Como os dados não são necessários em uma escrita, existem duas opções em uma falta na escrita:

- *Write allocate.* O bloco é alocado em uma falta na escrita, seguido das ações de acerto na escrita, acima. Nessa opção natural, as faltas de escrita atuam como faltas de leitura.
- *No-write allocate.* Nessa alternativa aparentemente incomum, as faltas de escrita *não* afetam a cache. Em vez disso, o bloco é modificado apenas na memória de nível inferior.

Assim, os blocos permanecem fora da cache no w*rite allocate* até que o programa tente ler os blocos, mas até mesmo os blocos que são apenas escritos ainda estarão na cache com a w*rite allocate*. Vejamos um exemplo.

Exemplo Considere uma cache write-back totalmente associativa com muitas entradas de cache que começam vazias. A seguir apresentamos uma sequência de cinco operações de memória (o endereço está entre colchetes):

```
Write Mem[100];
Write Mem[100];
Read Mem[200];
Write Mem[200];
Write Mem[100].
```

Quais são os números de acertos e faltas quando se usam a no-*write allocate* e a *write allocate*?

Resposta Para a no-*write allocate*, o endereço 100 não está na cache e não existe *write allocate*, de modo que as duas primeiras escritas resultarão em faltas. O endereço 200 também não está na cache, de modo que a leitura também é uma falta. A escrita subsequente no endereço 200 é um acerto. A última escrita em 100 ainda é uma falta. O resultado para a alocação sem escrita são quatro faltas e um acerto.

Para a alocação de escrita, os primeiros acessos a 100 e 200 são faltas, e o restante são acertos, pois 100 e 200 são encontrados na cache. Assim, o resultado para a alocação de escrita são duas faltas e três acertos.

Qualquer uma das políticas de falta de escrita poderia ser usada com write-through ou write-back. Normalmente, caches write-back utilizam *write allocate*, esperando que escritas subsequentes nesse bloco sejam capturadas pela cache. As caches write-through normalmente utilizam a no-*write allocate*. O motivo é que, mesmo se houver escritas subsequentes nesse bloco, as escritas ainda precisarão ir para a memória de nível inferior; logo, o que se ganha?

Um exemplo: a cache de dados do Opteron

Para dar fundo a essas ideias, a Figura B.5 mostra a organização da cache de dados no microprocessador AMD Opteron. A cache contém 65.536 (64K) bytes de dados em blocos de 64 bytes com alocação associativa por conjunto com duas vias, substituição pela usada menos recentemente, write-back e write allocate em uma falta na escrita.

Vamos rastrear um acerto na cache pelas etapas de um acerto, conforme rotulado na Figura B.5 (as quatro etapas são mostradas como números circulados). Conforme descrevemos na

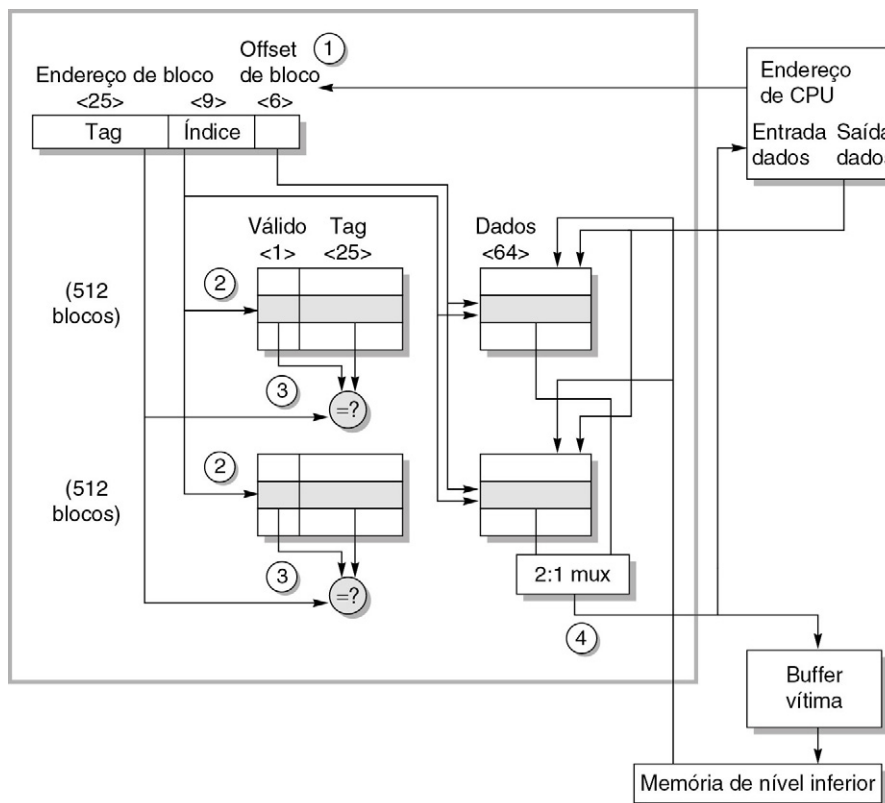



FIGURA B.5 **A organização da cache de dados no microprocessador Opteron.**
A cache de 64 KB é associativa por conjunto com duas vias, com blocos de 64 bytes. O índice de 9 bits seleciona entre 512 conjuntos. As quatro etapas de um acerto na leitura, mostradas como números circulados na ordem de ocorrência, rotulam essa organização. Três bits do offset de bloco se juntam ao índice para fornecer o endereço da RAM a fim de selecionar os 8 bytes apropriados. Assim, a cache mantém dois grupos de 4.096 palavras de 64 bits, com cada grupo contendo metade dos 512 conjuntos. Embora não ilustrado nesse exemplo, a fila da memória de nível inferior para a cache é usada em uma falta para carregar a cache. O tamanho do endereço saindo do processador tem 40 bits, pois é um endereço físico, e não um endereço virtual. A Figura B.23 explica como o Opteron mapeia da memória virtual para a física para um acesso à cache.

Seção B.5, o Opteron apresenta um endereço virtual de 48 bits para a cache para comparação de tag, que é simultaneamente traduzido para um endereço físico de 40 bits.

O motivo para o Opteron não usar todos os 64 bits do endereço virtual é que seus projetistas não acharam que alguém precisaria de um espaço de endereços virtuais tão grande, e o tamanho menor simplifica o mapeamento de endereços virtuais do Opteron. Os projetistas planejam aumentar o endereço virtual nos futuros microprocessadores.

O endereço físico vindo para a cache é dividido em dois campos: o endereço de bloco de 34 bits e o offset de bloco de 6 bits ($64 = 2^6$ e $34 + 6 = 40$). O endereço do bloco é dividido ainda em mais uma tag de endereço e índice de cache. A etapa 1 mostra essa divisão.

O índice de cache seleciona a tag a ser testada para saber se o bloco desejado está na cache. O tamanho do índice depende do tamanho da cache, do tamanho do bloco e da associatividade por conjunto. Para a cache do Opteron, a associatividade por conjunto é definida como dois, e calculamos o índice da seguinte forma:

$$2^{\text{Índice}} = \frac{\text{Tamanho de cache}}{\text{Tamanho de bloco} \times \text{Associatividade por conjunto}} = \frac{65,536}{64 \times 2} = 512 = 2^9$$

Logo, o índice tem 9 bits de largura, e a tag tem 34 – 9 ou 25 bits de largura. Embora esse seja o índice necessário para selecionar o bloco apropriado, 64 bytes é muito mais do que o processador deseja consumir ao mesmo tempo. Logo, faz mais sentido organizar a parte de dados da memória cache com 8 bytes de largura, que é uma palavra de dados natural do processador Opteron de 64 bits. Assim, além dos 9 bits para indexar o bloco de cache apropriado, mais 3 bits do offset de bloco são usados para indexar os 8 bytes apropriados. A seleção de índice é a etapa 2 da Figura B.5.

Depois de ler as duas tags da cache, elas são comparadas com a parte da tag do endereço do bloco do processador. Essa comparação é a etapa 3 da figura. Para ter certeza de que a tag contém informações válidas, o bit de validade precisa estar marcado ou os resultados da comparação serão ignorados.

Supondo que uma tag não seja igual, a última etapa é sinalizar o processador para carregar os dados apropriados da cache, usando a entrada escolhida de um multiplexador 2:1. O Opteron permite dois ciclos de clock para essas quatro etapas, de modo que as instruções nos dois ciclos de clock seguintes esperariam se tentassem usar o resultado do carregamento.

O tratamento das escritas é mais complicado do que o tratamento das leituras no Opteron, como em qualquer cache. Se a palavra a ser escrita estiver na cache, as três primeiras etapas serão iguais. Como o Opteron executa fora de ordem, somente depois de sinalizar que a instrução foi confirmada e a comparação da tag de cache indicar um acerto é que os dados serão escritos na cache.

Até aqui, consideramos o caso comum de um acerto na cache. O que acontece em uma falta? Em uma falta na leitura, a cache envia um sinal ao processador, dizendo que os dados ainda não estão disponíveis, e 64 bytes são lidos do próximo nível da hierarquia. A latência é de sete ciclos de clock para os oito primeiros bytes do bloco, e depois dois ciclos de clock por 8 bytes para o restante do bloco. Como a cache de dados é associativa por conjunto, existe uma escolha sobre qual bloco substituir. O Opteron usa LRU, que seleciona o bloco que foi referenciado há mais tempo, de modo que cada acesso precisa atualizar o bit LRU. Substituir o bloco significa atualizar os dados, a tag de endereço, o bit de validade e o bit LRU.

Como o Opteron utiliza write-back, o bloco de dados antigo poderia ter sido modificado e, portanto, não pode simplesmente ser descartado. O Opteron mantém 1 bit de modificação por bloco para registrar se o bloco foi escrito. Se a "vítima" foi modificada, seus dados e endereço são enviados ao buffer de vítima (essa estrutura é semelhante a um *buffer de escrita* em outros computadores). O Opteron tem espaço para oito blocos de vítima. Em paralelo com outras ações de cache, ele escreve os blocos de vítima no próximo nível da hierarquia. Se o buffer de vítima estiver cheio, a cache precisará esperar.

Uma falta na escrita é muito semelhante a uma falta na leitura, pois o Opteron aloca um bloco em uma falta na leitura ou na escrita.

Vimos como ele funciona, mas a cache de *dados* não pode fornecer todas as necessidades de memória do processador. O processador também precisa de instruções. Embora uma única cache pudesse tentar fornecer ambos, isso pode ser um gargalo. Por exemplo, quando uma instrução load ou store é executada, o processador com pipeline solicita simultaneamente a palavra de dados *e* uma palavra de instrução. Logo, uma única cache apresentaria um perigo estrutural para loads e stores, ocasionando stalls. Um modo simples de vencer esse problema é dividi-lo: uma cache é dedicada a instruções e outra aos dados. As caches separadas são encontradas na maioria dos processadores recentes, incluindo o Opteron. Logo, ele tem uma cache de instruções de 64 KB e também uma cache de dados de 64 KB.

O processador sabe se está enviando um endereço de instrução ou um endereço de dados, de modo que pode haver portas separadas para ambos, dobrando assim a largura de banda

Tamanho	Cache de instruções	Cache de dados	Cache unificada
8 KB	8,16	44,0	63,0
16 KB	3,82	40,9	51,0
32 KB	1,36	38,4	43,3
64 KB	0,61	36,9	39,4
128 KB	0,30	35,3	36,2
256 KB	0,02	32,6	32,9

FIGURA B.6 Falta por 1.000 instruções para caches de instruções, dados e unificada de tamanhos diferentes. A porcentagem de referências de instruções é de cerca de 74%. Os dados são para as caches associativas com duas vias, com blocos de 64 bytes para o mesmo computador e benchmarks, conforme a Figura B.4.

entre a hierarquia de memória e o processador. Caches separadas também oferecem a oportunidade de otimizar cada cache separadamente. Diferentes capacidades, tamanhos de bloco e associatividades podem levar a um desempenho melhor. (Ao contrário das caches de instruções e das caches de dados do Opteron, os termos *unificada* e *mista* são aplicados às caches, que podem conter instruções ou dados.)

A Figura B.6 mostra que as caches de instruções possuem menores taxas de falta do que as caches de dados. A separação entre instruções e dados remove as faltas decorrentes de conflitos entre blocos de instruções e blocos de dados, mas a divisão também fixa o espaço da cache dedicado a cada tipo. O que é mais importante para as taxas de falta? Uma comparação justa entre caches de instruções e dados separados e as caches unificadas requer que o tamanho total da cache seja o mesmo. Por exemplo, uma cache de instruções de 16 KB separada deve ser comparada com uma cache unificada de 32 KB. Para calcular a taxa de falta média com caches separadas para instruções e dados, é preciso saber a porcentagem de referências de memória a cada cache. A partir dos dados no Apêndice A, descobrimos que a divisão é de 100%/(100% + 26% + 10%) ou cerca de 74% de referências de instruções para (26% + 10%)/(100% + 26% + 10%) ou cerca de 26% de referências de dados. A divisão afeta o desempenho além do que é indicado pela mudança nas taxas de falta, conforme veremos em breve.

B.2 DESEMPENHO DE CACHE

Como o número de instruções é independente do hardware, é tentador avaliar o desempenho do processador usando esse número. Essas medidas de desempenho indiretas têm armado emboscadas para muitos projetistas de computador. A tentação correspondente para avaliar o desempenho da hierarquia de memória é concentrar-se na taxa de falta, porque ela também é independente da velocidade do hardware. Conforme veremos, a taxa de falta pode ser tão enganosa quanto o número de instruções. Uma medida melhor do desempenho da hierarquia de memória é o *tempo médio de acesso à memór*ia:

$$\text{Tempo médio de acesso à memória} = \text{Tempo de acerto} + \text{Taxa de falta} \times \text{Penalidade de falta}$$

onde o *tempo de acerto* é o tempo para acertar na cache; já vimos os dois outros termos. Os componentes do tempo médio de acesso podem ser medidos em tempo absoluto — por exemplo, 0,25 para 1,0 nanossegundo em um acerto — ou em número de ciclos de clock que o processador espera pela memória — como uma penalidade por falta de 150-200 ciclos de clock. Lembre-se de que o tempo médio de acesso à memória ainda é uma medida indireta do desempenho; embora sendo uma medida melhor do que a taxa de falta, isso não substitui o tempo de execução.

Essa fórmula pode nos ajudar a decidir entre caches divididas e uma cache unificada.

Exemplo Qual tem a menor taxa de falta: uma cache de instruções de 16 KB com uma cache de dados de 16 KB ou uma cache unificada de 32 KB? Use as taxas de falta da Figura B.6 para ajudar a calcular a resposta correta, considerando que 36% das instruções são instruções de transferência de dados. Considere que um acerto utiliza um ciclo de clock e a penalidade por falta é de 100 ciclos de clock. Um acerto no carregamento ou armazenamento utiliza um ciclo de clock extra em uma cache unificada se houver apenas uma porta de cache para atender a duas solicitações simultâneas. Usando a terminologia de pipelining do Capítulo 3, a cache unificada leva a um hazard estrutural. Qual é o tempo médio de acesso à memória em cada caso? Considere caches write-through com um buffer de escrita e ignore os stalls devidos ao buffer de escrita.

Resposta Primeiro, vamos converter as faltas por 1.000 instruções em taxas de falta. Resolvendo a fórmula geral apresentada, a taxa de falta é

$$\text{Taxa de falta} = \frac{\dfrac{\text{Faltas}}{1.000\ \text{instruções}} / 100}{\dfrac{\text{Acessos à memória}}{\text{Instrução}}}$$

Como cada acesso à instrução tem exatamente um acesso à memória para apanhar a instrução, a taxa de falta de instrução é

$$\text{Taxa de falta}_{\text{16KB de instruções}} = \frac{3{,}82 / 100}{1{,}00} = 0{,}004$$

Como 36% das instruções são transferências de dados, a taxa de falta de dados é

$$\text{Taxa de falta}_{\text{16KB de instruções}} = \frac{40{,}9 / 100}{0{,}36} = 0{,}114$$

A taxa de falta unificada precisa levar em consideração os acessos a instruções e dados:

$$\text{Taxa de falta}_{\text{32KB de instruções}} = \frac{43{,}3 / 100}{1{,}00 + 0{,}36} = 0{,}0318$$

Como já dissemos, cerca de 74% dos acessos à memória são referências a instruções. Assim, a taxa de falta geral para as caches divididas é de

$$(74\% \times 0{,}004) + (26\% \times 0{,}114) = 0{,}0326$$

Então, uma cache unificada de 32 KB possui uma taxa de falta efetiva ligeiramente menor do que duas caches de 16 KB.

A fórmula do tempo médio de acesso à memória pode ser dividida em acessos a instruções e dados:

$$\begin{aligned}&\text{Tempo médio de acesso à memória}\\&= \%\ \text{instruções} \times (\text{Tempo de acerto} + \text{Taxa de falta de instrução} \times \text{Penalidade de falta})\\&+ \%\ \text{dados}\ (\text{Tempo de acerto} + \text{Taxa de falta de dados} \times \text{Penalidade de falta})\end{aligned}$$

Portanto, o tempo para cada organização é

$$\begin{aligned}&\text{Tempo médio de acesso à memória}_{\text{dividida}}\\&= 74\% \times (1 + 0{,}004 \times 200) + 26\% \times (1 + 0{,}114 \times 200)\\&= (74\% \times 1{,}80) + (26\% \times 23{,}80) = 1{,}332 + 6{,}188 = 7{,}52\end{aligned}$$

$$\begin{aligned}&\text{Tempo médio de acesso à memória}_{\text{unificada}}\\&= 74\% \times (1 + 0{,}0318 \times 200) + 26\% \times (1 + 1 + 0{,}0318 \times 200)\\&= (74\% \times 7{,}36) + (26\% \times 8{,}36) = 5{,}446 + 2{,}174 = 7{,}62\end{aligned}$$

Logo, as caches divididas neste exemplo — que oferecem duas portas de memória por ciclo de clock, evitando assim o hazard estrutural — têm tempo médio de acesso à memória melhor do que a cache unificada de única porta, apesar de uma taxa de falta efetiva pior.

Tempo médio de acesso à memória e desempenho do processador

Uma questão óbvia é se o tempo médio de acesso à memória devido a faltas de cache prevê o desempenho do processador.

Primeiro, existem outros motivos para os stalls, como a disputa devida a dispositivos de E/S usando memória. Os projetistas normalmente consideram que todos os stalls da memória são devidos a faltas de cache, pois a hierarquia da memória normalmente domina outros motivos para stalls. Usamos essa suposição simplificada aqui, mas tenha o cuidado de levar em consideração *todos* os stalls de memória ao calcular o desempenho final.

Em segundo lugar, a resposta também depende do processador. Se tivermos um processador com execução em ordem (Cap. 3), então basicamente a resposta será sim. O processador atrasa durante as faltas, e o tempo de stall da memória é fortemente ligado ao tempo médio de acesso à memória. Vamos fazer essa suposição por enquanto, mas retornaremos aos processadores fora de ordem na próxima subseção.

Conforme indicamos na seção anterior, podemos modelar o tempo de CPU como

$$\text{Tempo de CPU} = (\text{Ciclos de clock de execução da CPU} + \text{Ciclos de clock de stall da memória}) \\ \times \text{Tempo do ciclo de clock}$$

Essa fórmula levanta a questão a respeito de os ciclos de clock para um acerto na cache deverem ser considerados parte dos ciclos de clock de execução da CPU ou parte dos ciclos de clock de stall da memória. Embora qualquer convenção possa ser defendida, a mais aceita é incluir os ciclos de clock de acerto nos ciclos de clock de execução da CPU.

Agora, podemos explorar o impacto das caches sobre o desempenho.

Exemplo Vamos usar um computador com execução em ordem para o primeiro exemplo. Considere que a penalidade por falta da cache seja 200 ciclos de clock e todas as instruções normalmente utilizem 1,0 ciclo de clock (ignorando os stalls de memória). Suponha que a taxa de falta média seja de 2%, que exista uma média de 1,5 referência de memória por instrução e que o número médio de faltas de cache por 1.000 instruções seja 30. Qual é o impacto no desempenho quando o comportamento da cache é incluído? Calcule o impacto usando faltas por instrução e a taxa de falta.

Resposta

$$\text{Tempo de CPU} = \text{IC} \times \left(\text{CPI}_{\text{execução}} + \frac{\text{Ciclos de stall da memória}}{\text{Instrução}} \right) \times \text{Tempo de ciclo de clock}$$

O desempenho, incluindo as faltas de cache, é

$$\begin{aligned} \text{Tempo de CPU}_{\text{com cache}} &= \text{IC} \times [1{,}0 + (30 / 1.000 \times 200)] \times \text{Tempo de ciclo de clock} \\ &= \text{IC} \times 7{,}00 \times \text{Tempo de ciclo de clock} \end{aligned}$$

Agora, calculando o desempenho usando a taxa de falta:

$$\text{Tempo de CPU} = \text{IC} \times \left(\text{CPI}_{\text{execução}} + \text{Taxa de perda} \times \frac{\text{Acessos à memória}}{\text{Instrução}} \right) \\ \times \text{Tempo de ciclo de clock}$$

$$\begin{aligned} \text{Tempo de CPU}_{\text{com cache}} &= \text{IC} \times [1{,}0 + (1{,}5 \times 2\% \times 200)] \times \text{Tempo de ciclo de clock} \\ &= \text{IC} \times 7{,}00 \times \text{Tempo de ciclo de clock} \end{aligned}$$

O tempo do ciclo de clock e o número de instruções são iguais, com ou sem cache. Assim, o tempo de CPU aumenta sete vezes, com CPI de 1,00 para uma "cache perfeita" a 7,00 com uma cache que pode ter faltas. Sem qualquer hierarquia de memória, o CPI aumentaria novamente para 1,0 + 200 × 1,5 ou 301 — um fator de mais de 40 vezes em relação a um sistema com uma cache!

Como esse exemplo ilustra, o comportamento da cache pode ter um impacto enorme sobre o desempenho. Além do mais, as faltas de cache possuem um impacto duplo sobre um processador com CPI baixo e clock rápido:

1. Quanto menor o $CPI_{execução}$, maior o impacto *relativo* de um número fixo de ciclos de clock na falta na cache.
2. Ao calcular o CPI, a penalidade por falta na cache é medida em ciclos de clock do processador para uma falta. Portanto, mesmo que as hierarquias de memória para dois computadores sejam idênticas, o processador com a taxa de clock mais alta tem número maior de ciclos de clock por falta e, daí, uma parte de memória mais alta do CPI.

A importância da cache para os processadores com CPI baixo e taxas de clock altas é, portanto, maior; consequentemente, maior é o perigo de desconsiderar o comportamento da cache na avaliação do desempenho de tais computadores. A lei de Amdahl ataca novamente!

Embora minimizar o tempo médio de acesso à memória seja um objetivo razoável — e usaremos isso em grande parte deste apêndice —, lembre-se de que o objetivo final é reduzir o tempo de execução do processador. O próximo exemplo mostra como esses dois podem diferir.

Exemplo Qual é o impacto de duas organizações de caches diferentes sobre o desempenho de um processador? Considere que o CPI com uma cache perfeita é de 1,6, o tempo de ciclo de clock é de 0,35 ns, existe 1,4 referência de memória por instrução, o tamanho das duas caches é de 128 KB e ambos possuem tamanho de bloco de 64 bytes. Uma cache é mapeada diretamente e a outra é associativa por conjunto com duas vias. A Figura B.5 mostra que, para caches associativas por conjunto, temos que acrescentar um multiplexador para selecionar entre os blocos no conjunto, dependendo do resultado da comparação das tags. Como a velocidade do processador pode estar ligada diretamente à velocidade de um acerto na cache, considere que o tempo de ciclo de clock do processador deva ser esticado 1,35 vez para acomodar o multiplexador de seleção da cache associativa por conjunto. Para a primeira aproximação, a penalidade por falta na cache é de 65 ns para qualquer organização de cache. (Na prática, normalmente ela é arredondada para cima ou para baixo, para um número inteiro de ciclos de clock.) Primeiro, calcule o tempo médio de acesso à memória e depois o desempenho do processador. Considere que o tempo de acerto seja de um ciclo de clock, a taxa de falta de uma cache de 128 KB mapeado diretamente seja de 2,1% e a taxa de falta para uma cache associativa por conjunto com duas vias do mesmo tamanho seja de 1,9%.

Resposta O tempo médio de acesso à memória é

$$\text{Tempo médio de acesso à memória} = \text{Tempo de acerto} + \text{Taxa de falta} \times \text{Penalidade de falta}$$

Assim, o tempo para cada organização é

$$\text{Tempo médio de acesso à memória}_{1\,\text{via}} = 0{,}35 + (0{,}021 \times 65) = 1{,}72 \text{ ns}$$
$$\text{Tempo médio de acesso à memória}_{2\,\text{vias}} = 0{,}35 \times 1{,}35 + (0{,}019 \times 65) = 1{,}71 \text{ ns}$$

O tempo médio de acesso à memória é melhor para a cache associativa por conjunto com duas vias.

O desempenho do processador é

$$\text{Tempo de CPU} = \text{IC} \times \left(\text{CPI}_{\text{execução}} + \frac{\text{Faltas}}{\text{Instrução}} \times \text{Penalidade de falta}\right) \times \text{Tempo de ciclo de clock}$$
$$= \text{IC} \times \Big[\left(\text{CPI}_{\text{execução}} \times \text{Tempo de ciclo de clock}\right)$$
$$+ \left(\text{Taxa de faltas} \times \frac{\text{Acessos à memória}}{\text{Instrução}} \times \text{Penalidade de falta} \times \text{Tempo de ciclo de clock}\right)\Big]$$

Substituindo (Penalidade por falta × Tempo de ciclo de clock) por 65 ns, o desempenho de cada organização de cache é

$$\text{Tempo de CPU}_{\text{1 via}} = \text{IC}\,(1{,}6 \times 0{,}35 + (0{,}021 \times 1{,}4 \times 65)) = 2{,}47 \times \text{IC}$$

$$\text{Tempo de CPU}_{\text{2 vias}} = \text{IC}\,(1{,}6 \times 0{,}35 \times 1{,}35(0{,}019 \times 1{,}4 \times 65)) = 2{,}49 \times \text{IC}$$

e o desempenho relativo é

$$\frac{\text{Tempo de CPU}_{\text{2 vias}}}{\text{Tempo de CPU}_{\text{1 via}}} = \frac{2{,}49 \times \text{Número de instruções}}{2{,}47 \times \text{Número de instruções}} = \frac{2{,}49}{2{,}47} = 1{,}01$$

Ao contrário dos resultados da comparação do tempo médio de acesso à memória, a cache com mapeamento direto leva a um desempenho ligeiramente melhor, pois o ciclo de clock é esticado para *todas* as instruções para o caso associativo por conjunto com duas vias, mesmo que existam menos faltas. Como o tempo de CPU é nossa avaliação final e como o mapeamento direto é mais simples de se montar, a cache preferida é mapeada diretamente neste exemplo.

Penalidade por falta e processadores com execução fora de ordem

Para um processador com execução fora de ordem, como você define a "penalidade por falta"? Ela é a latência completa da falta para a memória ou apenas a latência "exposta" ou não sobreposta quando o processador precisar gerar um stall? Essa questão não surge nos processadores que não geram antes que a falta nos dados termine.

Vamos redefinir os stalls da memória para levar a uma nova definição de penalidade por falta como latência não sobreposta:

$$\frac{\text{Ciclos de stall da memória}}{\text{Instrução}} = \frac{\text{Faltas}}{\text{Instrução}} \times (\text{Latência de falta total} - \text{latência de falta sobreposta})$$

De modo semelhante, como alguns processadores com execução fora de ordem esticam o tempo de acerto, essa parte da equação de desempenho poderia ser dividida pela latência de acerto total menos a latência de acerto sobreposta. Essa equação poderia ser expandida ainda mais para levar em consideração a disputa pelos recursos de memória em um processador fora de ordem, dividindo a latência de falta total pela latência sem disputa e a latência devida à disputa. Vamos nos concentrar apenas na latência de falta.

Agora, temos que decidir o seguinte:

- *Tamanho da latência da memória.* O que considerar como início e final de uma operação de memória em um processador fora de ordem.
- *Tamanho da sobreposição de latência.* Qual é o início da sobreposição com o processador (ou, de forma equivalente, quando dizemos que uma operação com a memória está fazendo com que o processador fique em stall)?

Dada a complexidade dos processadores com execução fora de ordem, não existe uma única definição correta.

Como somente as operações confirmadas são vistas no estágio de retirada da pipeline, dizemos que um processador ficará em stall em um ciclo de clock se não retirar o número máximo possível de instruções nesse ciclo. Atribuímos esse stall à primeira instrução que poderia não ser retirada. Essa definição não é infalível, de forma alguma. Por exemplo, aplicar uma otimização para melhorar certo tempo de stall pode nem sempre melhorar o tempo de execução, pois outro tipo de stall — escondido por trás do stall visado — pode estar exposto agora.

Para a latência, poderíamos começar medindo a partir do momento em que a instrução da memória é enfileirada na janela de instruções ou quando o endereço é gerado ou quando a instrução é realmente enviada para o sistema de memória. Qualquer opção funciona desde que seja usada de forma coerente.

Exemplo Vamos refazer o exemplo anterior, mas desta vez consideraremos que o processador com o maior tempo de ciclo de clock aceita a execução fora de ordem, mesmo ainda tendo uma cache com mapeamento direto. Considere que 30% da penalidade por falta de 65 ns podem ser sobrepostos; ou seja, o tempo médio de stall de memória da CPU agora é de 45,5 ns.

Resposta O tempo médio de acesso à memória para o computador fora de ordem é

$$\text{Tempo médio de acesso à memória}_{1\text{ via,OOO}} = 0{,}35 \times 1{,}35 + (0{,}021 \times 45{,}5) = 1{,}43 \text{ ns}$$

O desempenho da cache OOO é

$$\text{Tempo CPU}_{1\text{ via,OOO}} = \text{IC} \times (1{,}6 \times 0{,}35 \times 1{,}35 + (0{,}021 \times 1{,}4 \times 45{,}5)) = 2{,}09 \times \text{IC}$$

Logo, apesar de um tempo de ciclo de clock muito mais lento e da taxa de falta mais alta de uma cache mapeada diretamente, o computador fora de ordem pode ser ligeiramente mais rápido se puder ocultar 30% da penalidade por falta.

Resumindo, embora o estado da arte na definição e medição de stalls de memória para processadores fora de ordem seja complexo, esteja ciente dos problemas, porque eles afetam significativamente o desempenho. A complexidade cresce porque o procesador fora de ordem tolera algumas latências devido a faltas de caches sem prejudicar o desempenho. Consequentemente, os projetistas normalmente utilizam simuladores de processador fora de ordem e de memória na avaliação das escolhas na hierarquia da memória, para ter certeza de que uma melhoria que ajuda a latência média da memória realmente ajuda no desempenho do programa.

Para ajudar a resumir esta seção e atuar como uma referência prática, a Figura B.7 lista as equações de cache deste apêndice.

$$2^{\text{Índice}} = \frac{\text{Tamanho da cache}}{\text{Tamanho de bloco} \times \text{Associatividade por conjunto}} = \frac{65{,}536}{64 \times 2} = 512 = 2^9$$

$$\text{Tempo de execução da CPU} = (\text{Ciclos de clock da CPU} + \text{Ciclos de stall da memória}) \times \text{Tempo do ciclo de clock}$$

$$\text{Ciclos de stall da memória} = \text{Número de faltas} \times \text{Penalidade de falta}$$

$$\text{Ciclos de stall da memória} = \text{IC} \times \frac{\text{Faltas}}{\text{Instrução}} \times \text{Penalidade de falta}$$

$$\frac{\text{Faltas}}{\text{Instrução}} = \text{Taxa de falta} \times \frac{\text{Acesso à memória}}{\text{Instrução}}$$

$$\text{Tempo médio de acesso à memória} = \text{Tempo de acerto} + \text{Taxa de falta} \times \text{Penalidade de falta}$$

$$\text{Tempo de execução de CPU} = \left(\text{CPI}_{\text{execução}} + \frac{\text{Stall de memória}}{\text{Instrução}}\right) \times \text{Tempo de ciclo de clock}$$

$$\text{Tempo de execução de CPU} = \left(\text{CPI}_{\text{execução}} + \frac{\text{Faltas}}{\text{Instrução}} \times \text{Penalidade de falta}\right) \times \text{Tempo de ciclo de clock}$$

$$\text{Tempo de execução de CPU} = \left(\text{CPI}_{\text{execução}} + \text{Taxa de falta} \times \frac{\text{Acesso à memória}}{\text{Instrução}} \times \text{Penalidade de falta}\right) \times \text{Tempo de ciclo de clock}$$

$$\frac{\text{Ciclos de stall da memória}}{\text{Instrução}} = \frac{\text{Faltas}}{\text{Instrução}} \times (\text{Latência de falta total - latência de falta sobreposta})$$

$$\text{Tempo médio de acesso à memória} = \text{Tempo de acerto}_{L1} + \text{Taxa de falta}_{L1} \times (\text{Tempo de acerto}_{L2} + \text{Taxa de falta}_{L2} \times \text{Penalidade de falta}_{L2})$$

$$\frac{\text{Ciclos de stall da memória}}{\text{Instrução}} = \frac{\text{Faltas}_{L1}}{\text{Instrução}} \times \text{Tempo de acerto}_{L2} + \frac{\text{Faltas}_{L2}}{\text{Instrução}} \times \text{Penalidade de falta}_{L2}$$

FIGURA B.7 Resumo das equações de desempenho deste apêndice.
A primeira equação calcula o tamanho do índice de cache, e o restante ajuda na avaliação do desempenho. As duas últimas equações tratam das caches multiníveis, que serão explicadas no início da próxima seção. Elas foram incluídas aqui para que a figura possa se tornar uma referência útil.

B.3 SEIS OTIMIZAÇÕES DE CACHE BÁSICAS

A fórmula do tempo médio de acesso nos deu uma estrutura para apresentar otimizações de cache para melhorar o desempenho da cache:

$$\text{Tempo médio de acesso à memória} = \text{Tempo de acerto} + \text{Taxa de acerto} \times \text{Penalidade por falta}$$

Daí, organizamos as seis otimizações da cache em três categorias:

- *Reduzir a taxa de falta* — tamanho de bloco maior, tamanho de cache maior e associatividade mais alta
- *Reduzir a penalidade por falta* — caches multiníveis e dar prioridade às leituras em vez das escritas
- *Reduzir o tempo para acerto na cache* — evitar tradução de endereço ao indexar a cache

A Figura B.18, na página B-36, conclui esta seção com um resumo da complexidade da implementação e os benefícios de desempenho dessas seis técnicas.

A técnica clássica para melhorar o comportamento da cache é reduzir as taxas de falta, e apresentamos três técnicas para fazer isso. Para ter uma ideia melhor das causas de faltas, primeiro começamos com um modelo que classifica todas as faltas em três categorias simples:

- *Compulsória.* O primeiro acesso a um bloco *não pode* ser feito na cache, por isso o bloco precisa ser trazido para a cache. Essas também são chamadas *faltas de partida a frio* ou *faltas de primeira referência.*
- *Capacidade.* Se a cache não puder conter todos os blocos necessários durante a execução de um programa, as faltas de capacidade (além das faltas compulsórias) ocorrerão por causa dos blocos sendo descartados e recuperados mais tarde.
- *Conflito.* Se a estratégia de colocação do bloco é a associatividade em conjunto ou o mapeamento direto, faltas por conflito (além das faltas compulsórias e por capacidade) ocorrerão porque um bloco pode ser descartado e mais tarde recuperado se muitos blocos forem mapeados para o seu conjunto. Essas faltas também são chamadas *faltas por colisão*. A ideia é que os acertos em uma cache totalmente associativa, que se tornam faltas em uma cache associativa em conjunto com *n* vias, devem-se a mais do que *n* solicitações em alguns conjuntos populares.

(O Capítulo 5 acrescenta um quarto C, para faltas de *coerência*, decorrentes de esvaziamentos de cache para manter várias caches coerentes em um multiprocessador; não vamos considerá-las aqui.)

A Figura B.8 mostra a frequência relativa das faltas de cache, desmembradas pelos "três C". As faltas compulsórias são aquelas que ocorrem em uma cache infinita. As faltas por capacidade são aquelas que ocorrem em uma cache totalmente associativa. As faltas por conflito são aquelas que ocorrem de uma cache totalmente associativa para uma associativa com oito vias, associativa com quatro vias, e assim por diante. A Figura B.9 apresenta os mesmos dados graficamente. O gráfico superior mostra as taxas de falta absolutas; o gráfico inferior desenha a porcentagem de todas as faltas por tipo de falta como uma função do tamanho da cache.

Para mostrar o benefício da associatividade, as faltas por conflito são divididas em faltas causadas por cada diminuição na associatividade. Aqui estão as quatro divisões das faltas por conflito e como elas são calculadas:

- *Oito vias.* Faltas por conflito devidas à passagem de totalmente associativa (sem conflitos) para associativa com oito vias.
- *Quatro vias.* Faltas por conflito devidas à passagem de associativa com oito vias para associativa com quatro vias.
- *Duas vias.* Faltas de conflito devidas à passagem de associativa com quatro vias para associativa com duas vias.
- *Uma via.* Faltas de conflito devidas à passagem de associativa com duas vias para associativa com uma via (mapeamento direto).

Como podemos ver pelas figuras, a taxa de falta compulsória dos programas SPEC2000 é muito pequena, assim como para muitos programas de longa duração.

Tendo identificado os três C, o que um projetista de computador pode fazer com eles? Por conceito, os conflitos são os mais fáceis: o mapeamento totalmente associativo evita todas as faltas por conflito. Porém, a associatividade total é dispendiosa em hardware e pode atrasar a taxa de clock do processador (veja o exemplo na página B-29), levando a um desempenho geral mais baixo.

Tamanho da cache (KB)	Grau de associatividade	Taxa de falta total	Componentes da taxa de falta (porcentagem relativa) (*soma = 100% da taxa de falta total*)					
			Compulsória		Capacidade		Conflito	
4	1 via	0,098	0,0001	0,1%	0,070	72%	0,027	28%
4	2 vias	0,076	0,0001	0,1%	0,070	93%	0,005	7%
4	4 vias	0,071	0,0001	0,1%	0,070	99%	0,001	1%
4	8 vias	0,071	0,0001	0,1%	0,070	100%	0,000	0%
8	1 via	0,068	0,0001	0,1%	0,044	65%	0,024	35%
8	2 vias	0,049	0,0001	0,1%	0,044	90%	0,005	10%
8	4 vias	0,044	0,0001	0,1%	0,044	99%	0,000	1%
8	8 vias	0,044	0,0001	0,1%	0,044	100%	0,000	0%
16	1 via	0,049	0,0001	0,1%	0,040	82%	0,009	17%
16	2 vias	0,041	0,0001	0,2%	0,040	98%	0,001	2%
16	4 vias	0,041	0,0001	0,2%	0,040	99%	0,000	0%
16	8 vias	0,041	0,0001	0,2%	0,040	100%	0,000	0%
32	1 via	0,042	0,0001	0,2%	0,037	89%	0,005	11%
32	2 vias	0,038	0,0001	0,2%	0,037	99%	0,000	0%
32	4 vias	0,037	0,0001	0,2%	0,037	100%	0,000	0%
32	8 vias	0,037	0,0001	0,2%	0,037	100%	0,000	0%
64	1 via	0,037	0,0001	0,2%	0,028	77%	0,008	23%
64	2 vias	0,031	0,0001	0,2%	0,028	91%	0,003	9%
64	4 vias	0,030	0,0001	0,2%	0,028	95%	0,001	4%
64	8 vias	0,029	0,0001	0,2%	0,028	97%	0,001	2%
128	1 via	0,021	0,0001	0,3%	0,019	91%	0,002	8%
128	2 vias	0,019	0,0001	0,3%	0,019	100%	0,000	0%
128	4 vias	0,019	0,0001	0,3%	0,019	100%	0,000	0%
128	8 vias	0,019	0,0001	0,3%	0,019	100%	0,000	0%
256	1 via	0,013	0,0001	0,5%	0,012	94%	0,001	6%
256	2 vias	0,012	0,0001	0,5%	0,012	99%	0,000	0%
256	4 vias	0,012	0,0001	0,5%	0,012	99%	0,000	0%
256	8 vias	0,012	0,0001	0,5%	0,012	99%	0,000	0%
512	1 via	0,008	0,0001	0,8%	0,005	66%	0,003	33%
512	2 vias	0,007	0,0001	0,9%	0,005	71%	0,002	28%
512	4 vias	0,006	0,0001	1,1%	0,005	91%	0,000	8%
512	8 vias	0,006	0,0001	1,1%	0,005	95%	0,000	4%

FIGURA B.8 Taxa de falta total para cada tamanho de cache e porcentagem de cada um de acordo com os "três C".
As faltas compulsórias são independentes do tamanho da cache, enquanto as faltas por capacidade diminuem à medida que a capacidade aumenta, e as faltas por conflito diminuem à medida que a associatividade aumenta. A Figura B.9 mostra a mesma informação graficamente. Observe que uma cache mapeada diretamente com tamanho N tem aproximadamente a mesma taxa de falta em uma cache associativa em conjunto com duas vias, de tamanho N/2, até 128 K. As caches maiores que 128 KB não provam essa regra. Observe que a coluna Capacidade também é a taxa de falta totalmente associativa. Os dados foram coletados como na Figura B.4, usando a substituição LRU.

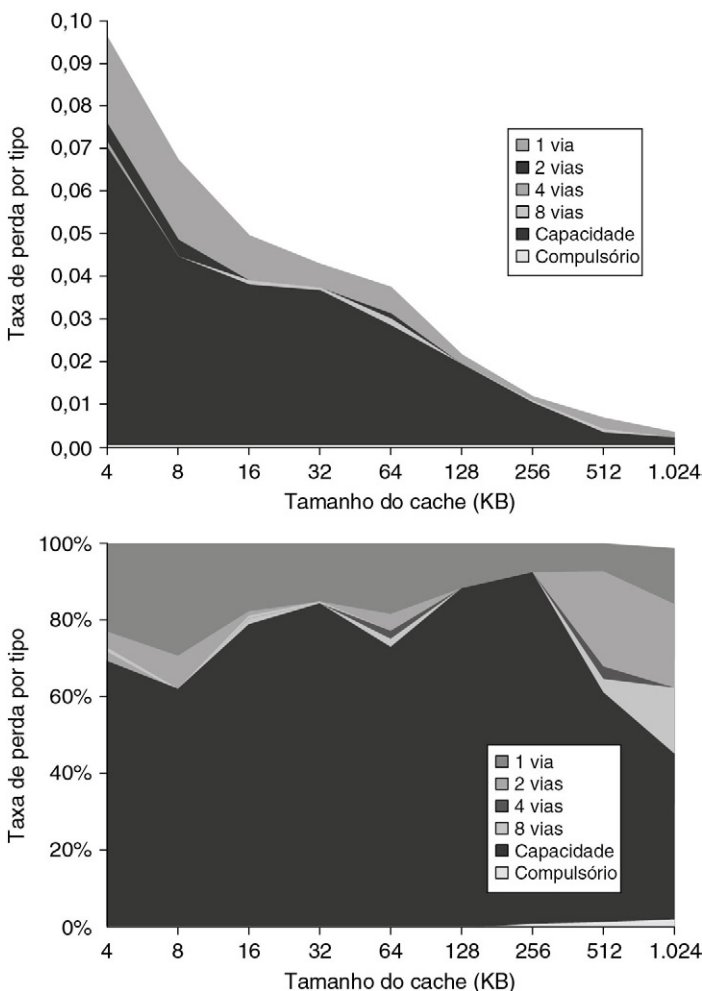



FIGURA B.9 **A taxa de falta total (superior) e a distribuição da taxa de falta (inferior) para cada tamanho de cache de acordo com os três C para os dados da Figura B.8.**
O diagrama superior são as taxas de falta na cache com dados reais, enquanto o diagrama inferior mostra a porcentagem em cada categoria. (O espaço permite que os gráficos mostrem um tamanho de cache extra, além do que pode caber na Figura B.8.)

Existe pouca coisa a ser feita a respeito da capacidade, exceto ampliar a cache. Se a memória de nível superior for muito menor do que a necessária para um programa e uma porcentagem significativa do tempo for gasta movendo dados entre dois níveis na hierarquia, a hierarquia de memória será considerada *thrashing*. Como são exigidas muitas substituições, o thrashing significa que o computador trabalha perto da velocidade da memória de nível inferior ou, talvez, até menos que isso, devido ao overhead da falta.

Outra técnica para melhorar os três C é tornar os blocos maiores, para reduzir o número de faltas compulsórias, mas, como veremos em breve, blocos grandes podem aumentar outros tipos de faltas.

Os três C dão uma ideia da causa das faltas, mas esse modelo simples tem seus limites; ele lhe dá uma ideia do comportamento médio, mas não explica uma falta individual. Por exemplo, alterar o tamanho da cache muda as faltas por conflito e também as faltas por capacidade, pois uma cache maior espalha referências a mais blocos. Assim, uma

falta poderia passar de uma falta na capacidade para uma falta por conflito, enquanto o tamanho da cache muda. Observe que os três C também ignoram a política de substituição, pois ela é difícil de modelar e porque, em geral, é menos significativa. Em circunstâncias específicas, a política de substituição pode realmente levar a um comportamento anômalo, como taxas de falta mais fracas para associatividade maior, o que contradiz o modelo dos três C. (Alguns propuseram usar um rastreio de endereço para determinar o posicionamento ideal na memória, para evitar faltas de posicionamento do modelo dos três C; não seguimos esse conselho aqui.)

Infelizmente, muitas das técnicas que reduzem as taxas de falta também aumentam o tempo de acerto ou a penalidade de falta. O desejo de reduzir as taxas de falta usando as três otimizações precisa ser balanceado com o objetivo de tornar o sistema mais rápido como um todo. Esse primeiro exemplo mostra a importância de uma perspectiva balanceada.

Primeira otimização: tamanho de bloco maior para reduzir a taxa de falta

O modo mais simples de reduzir a taxa de falta é aumentar o tamanho do bloco. A Figura B.10 mostra a escolha entre tamanho de bloco e taxa de falta para um conjunto de programas e tamanhos de cache. Tamanhos de bloco maiores reduzirão também as faltas compulsórias. Essa redução ocorre porque o princípio da localidade tem dois componentes: localidade temporal e localidade espacial. Blocos maiores tiram proveito da localidade espacial.

Ao mesmo tempo, blocos maiores aumentam a penalidade por falta. Como eles reduzem o número de blocos na cache, blocos maiores podem aumentar as faltas por conflito e até mesmo as faltas por capacidade se a cache for pequena. Nitidamente, existe pouco motivo para aumentar o tamanho do bloco para um tamanho tal que *aumente* a taxa de falta. Também não existe benefício em reduzir a taxa de falta se isso aumentar o tempo médio de acesso à memória. O aumento na penalidade por falta pode ser superior à diminuição na taxa de falta.

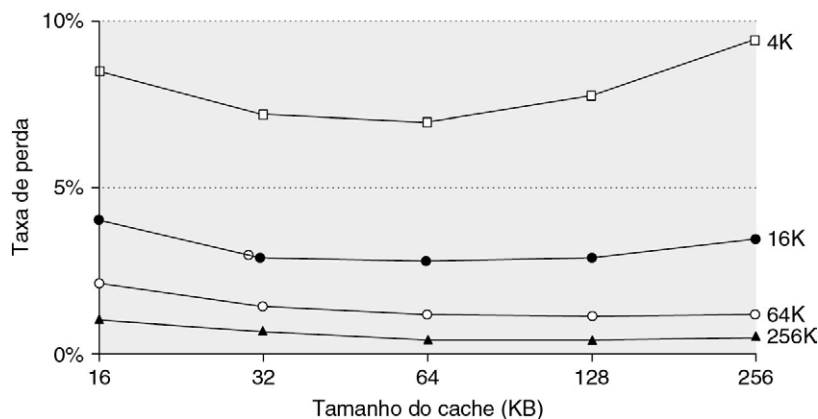



FIGURA B.10 **Taxa de falta *versus* tamanho de bloco para cinco caches de tamanhos diferentes.**
Observe que a taxa de falta realmente sobe quando o tamanho do bloco for muito grande em relação ao tamanho da cache. Cada linha representa uma cache de tamanho diferente. A Figura C.11 mostra os dados usados para desenhar essas linhas. Infelizmente, os registros do SPEC2000 seriam muito longos se o tamanho do bloco fosse incluído, de modo que esses dados são baseados no SPEC92 em um DECstation 5000 (Gee et al., 1993).

Exemplo A Figura B.11 mostra as taxas de falta reais desenhadas na Figura B.10. Considere que o sistema de memória usa 80 ciclos de clock de overhead e depois entrega 16 bytes a cada dois ciclos de clock. Assim, ele pode fornecer 16 bytes em 82 ciclos de clock, 32 bytes em 84 ciclos de clock, e assim por diante. Qual tamanho de bloco tem o menor tempo médio de acesso à memória para cada tamanho de cache da Figura B.11?

Resposta O tempo médio de acesso à memória é

$$\text{Tempo médio de acesso à memória} = \text{Tempo de acerto} + \text{Taxa de falta} \times \text{Penalidade por falta}$$

Se considerarmos que o tempo de acerto é um ciclo de clock, independentemente do tamanho do bloco, então o tempo de acesso para um bloco de 16 bytes em uma cache de 4 KB é

$$\text{Tempo médio de acesso à memória} = 1 + (8{,}57\% \times 82) = 8{,}027 \text{ ciclos de clock}$$

e para um bloco de 256 bytes em uma cache de 256 KB, o tempo médio de acesso à memória é

$$\text{Tempo médio de acesso à memória} = 1 + (0{,}49\% \times 112) = 1{,}549 \text{ ciclo de clock}$$

A Figura B.12 mostra o tempo médio de acesso à memória para todos os tamanhos de bloco e cache entre esses dois extremos. As entradas em negrito mostram o tamanho de bloco mais rápido para determinado tamanho de cache: 32 bytes para 4 KB e 64 bytes para as caches maiores. Esses tamanhos, na verdade, são referentes a bloco populares para as caches de processador hoje em dia.

	Tamanho da cache			
Tamanho de bloco	**4 K**	**16 K**	**64 K**	**256 K**
16	8,57%	3,94%	2,04%	1,09%
32	7,24%	2,87%	1,35%	0,70%
64	7,00%	2,64%	1,06%	0,51%
128	7,78%	2,77%	1,02%	0,49%
256	9,51%	3,29%	1,15%	0,49%

FIGURA B.11 Taxa de falta real contra tamanho de bloco para cinco caches de tamanhos diferentes na Figura B.10.
Observe que, para uma cache de 4 KB, blocos de 256 bytes possuem uma taxa de falta mais alta do que os blocos de 32 bytes. Nesse exemplo, a cache teria que ser de 256 KB para que um bloco de 256 bytes diminuísse as faltas.

Como em todas essas técnicas, o projetista de cache está tentando minimizar a taxa de falta e a penalidade por falta. A seleção de tamanho de bloco depende tanto da latência quanto da largura de banda da memória de nível inferior. Alta latência e alta largura de banda encorajam o grande tamanho de bloco, pois a cache recebe muito mais bytes por falta para um pequeno aumento na penalidade de falta. Ao contrário, baixa latência e baixa largura de banda encorajam menores tamanhos de bloco, pois pouco tempo é economizado com um bloco maior. Por exemplo, o dobro da penalidade por falta de um bloco pequeno pode ser próximo da penalidade de um bloco com o dobro do tamanho. O número maior de blocos pequenos também pode reduzir as faltas por conflito. Observe que as Figuras B.10 e B.12 mostram a diferença entre selecionar um tamanho de bloco com base na redução da taxa de falta contra a redução do tempo médio de acesso à memória.

Tamanho de bloco	Penalidade de falta	Tamanho da cache			
		4 K	16 K	64 K	256 K
16	82	8.027	4.231	2.673	1.894
32	84	**7.082**	3.411	2.134	1.588
64	88	7.160	**3.323**	**1.933**	**1.449**
128	96	8.469	3.659	1.979	1.470
256	112	11.651	4.685	2.288	1.549

FIGURA B.12 Tempo médio de acesso à memória contra tamanho de bloco para cinco caches de tamanhos diferentes na Figura B.10.

Os tamanhos de bloco de 32 e 64 bytes dominam. O tempo médio menor por tamanho de cache está em negrito.

Depois de ver os impactos positivo e negativo do tamanho de bloco maior nas faltas compulsórias e por capacidade, as duas subseções seguintes examinam o potencial da maior capacidade e da maior associatividade.

Segunda otimização: caches maiores para reduzir a taxa de falta

O modo óbvio de reduzir as faltas por capacidade nas Figuras B.8 e B.9 é aumentar a capacidade da cache. A desvantagem óbvia é o tempo de acerto potencialmente maior, além de maior custo e maior potência. Essa técnica tem sido popular especialmente nas caches fora do chip.

Terceira otimização: associatividade maior para reduzir a taxa de falta

As Figuras B.8 e B.9 mostram como as taxas de falta melhoram com a associatividade maior. Existem duas regras práticas gerais que podem ser concluídas dessas figuras. A primeira é que a associatividade por conjunto com oito vias é, para fins práticos, tão eficiente na redução de faltas para esses tamanhos de cache quanto a associatividade total. Você pode ver a diferença comparando as entradas de oito vias com a coluna de falta por capacidade na Figura B.8, pois as faltas por capacidade são calculadas usando caches totalmente associativas.

A segunda observação, chamada *regra prática de cache 2:1*, é que uma cache com mapeamento direto de tamanho *N* tem aproximadamente a mesma taxa de falta de uma cache associativa por conjunto com duas vias, com tamanho *N*/2. Isso se manteve nas figuras dos três C para os tamanhos de cache menores que 128 KB.

Como em muitos desses exemplos, melhorar um aspecto do tempo médio de acesso à memória custará outro aspecto. Aumentar o tamanho do bloco reduz a taxa de falta, enquanto aumenta a penalidade por falta, e uma associatividade maior pode ter o custo de um aumento no tempo de acerto. Logo, a pressão de um ciclo de clock de um processador rápido encoraja projetos de cache simples, mas o aumento da penalidade por falta recompensa a associatividade, como sugere o exemplo a seguir.

Exemplo Considere que a associatividade maior aumentaria o tempo do ciclo de clock, conforme listado a seguir:

Tempo de ciclo de $\text{clock}_{2 \text{ vias}} = 1{,}36 \times$ Tempo de ciclo de $\text{clock}_{1 \text{ via}}$
Tempo de ciclo de $\text{clock}_{4 \text{ vias}} = 1{,}44 \times$ Tempo de ciclo de $\text{clock}_{1 \text{ via}}$
Tempo de ciclo de $\text{clock}_{8 \text{ vias}} = 1{,}52 \times$ Tempo de ciclo de $\text{clock}_{1 \text{ via}}$

Considere que o tempo de acerto seja de um ciclo de clock, que a penalidade por falta para o caso de mapeamento direto seja de 25 ciclos de clock para uma cache nível 2 (ver a próxima subseção), que nunca ocorre falta e que a penalidade por falta não precise ser arredondada para um número inteiro de ciclos de clock. Usando a Figura B.8 para as taxas de falta, para quais tamanhos de cache cada uma dessas três afirmações é verdadeira?

Tempo médio de acesso à $\text{memória}_{8 \text{ vias}} <$ Tempo médio de acesso à $\text{memória}_{4 \text{ vias}}$
Tempo médio de acesso à $\text{memória}_{4 \text{ vias}} <$ Tempo médio de acesso à $\text{memória}_{2 \text{ vias}}$
Tempo médio de acesso à $\text{memória}_{2 \text{ vias}} <$ Tempo médio de acesso à $\text{memória}_{1 \text{ via}}$

Resposta O tempo médio de acesso à memória para cada associatividade é

Tempo médio de acesso à $\text{memória}_{8 \text{ vias}} =$ Tempo de $\text{acerto}_{8 \text{ vias}} +$ Taxa de $\text{falta}_{8 \text{ vias}} \times$ Penalidade por $\text{falta}_{8 \text{ vias}} = 1{,}52 +$ Taxa de $\text{falta}_{8 \text{ vias}} \times 25$
Tempo médio de acesso à $\text{memória}_{4 \text{ vias}} = 1{,}44 +$ Taxa de $\text{falta}_{4 \text{ vias}} \times 25$
Tempo médio de acesso à $\text{memória}_{2 \text{ vias}} = 1{,}36 +$ Taxa de $\text{falta}_{2 \text{ vias}} \times 25$
Tempo médio de acesso à $\text{memória}_{1 \text{ via}} = 1{,}00 +$ Taxa de $\text{falta}_{1 \text{ vias}} \times 25$

A penalidade por falta tem o mesmo tempo em cada caso, de modo que a deixamos como 25 ciclos de clock. Por exemplo, o tempo médio de acesso à memória para uma cache de 4 KB mapeado diretamente é

Tempo médio de acesso à $\text{memória}_{1 \text{ via}} = 1{,}00 + (0{,}098 \times 25) = 3{,}44$

e o tempo para uma cache associativa em conjunto com oito vias é

Tempo médio de acesso à $\text{memória}_{8 \text{ vias}} = 1{,}52 + (0{,}006 \times 25) = 1{,}66$

Usando essas fórmulas e as taxas de falta da Figura B.8, a Figura B.13 mostra o tempo médio de acesso à memória para cada cache e para cada associatividade. A figura mostra que as fórmulas nesse exemplo se mantêm para caches menores ou iguais a 8 KB para uma associatividade de até quatro vias. A partir de 16 KB, o maior tempo de acerto da associatividade maior é superior ao tempo economizado devido à redução nas faltas.
Observe que não levamos em consideração a taxa de clock mais lenta no restante do programa neste exemplo, subestimando assim a vantagem da cache mapeada diretamente.

Quarta otimização: caches multiníveis para reduzir a penalidade por falta

Reduzir as faltas na cache tem sido o foco tradicional da pesquisa da cache, mas a fórmula de desempenho da cache nos garante que as melhorias na penalidade por falta podem ser tão benéficas quanto as melhorias na taxa de falta. Além do mais, a Figura 2.2, na página 63, mostra que as tendências da tecnologia tornaram a velocidade dos processadores maior que a das DRAMs, fazendo com que o custo relativo das penalidades por falta aumente com o tempo.

Essa lacuna de desempenho entre os processadores e a memória leva o arquiteto a esta pergunta: devo tornar a cache mais rápida para acompanhar a velocidade dos processadores ou tornar a cache maior para contornar a maior lacuna entre o processador e a memória principal?

	Associatividade			
Tamanho da cache (KB)	**Uma via**	**Duas vias**	**Quatro vias**	**Oito vias**
4	3,44	3,25	3,22	**3,28**
8	2,69	**2,58**	**2,55**	**2,62**
16	2,23	**2,40**	**2,46**	**2,53**
32	2,06	**2,30**	**2,37**	**2,45**
64	1,92	**2,14**	**2,18**	**2,25**
128	1,52	**1,84**	**1,92**	**2,00**
256	1,32	**1,66**	**1,74**	**1,82**
512	1,20	**1,55**	**1,59**	**1,66**

FIGURA B.13 Tempo médio de acesso à memória usando taxas de falta da Figura B.8 para os parâmetros no exemplo.
O tipo em negrito significa que esse tempo é maior que o número à esquerda; ou seja, maior associatividade aumenta o tempo médio de acesso à memória.

Uma resposta é: faça ambos. Incluir outro nível de cache entre a cache original e a memória simplifica a decisão. A cache de primeiro nível pode ser pequena o suficiente para corresponder ao tempo de ciclo de clock do processador veloz. Porém, a cache de segundo nível pode ser grande o suficiente para capturar muitos acessos que iriam para a memória principal, reduzindo assim a penalidade por falta efetiva.

Embora o conceito de acrescentar outro nível na hierarquia seja simples, isso complica a análise de desempenho. As definições para um segundo nível de cache nem sempre são simples. Vamos começar com a definição de *tempo médio de acesso à memória* para uma cache de nível dois. Usando os subscritos L1 e L2 para nos referir, respectivamente, a uma cache de primeiro e segundo nível, a fórmula original é

$$\text{Tempo médio de acesso à memória} = \text{Tempo de acerto}_{L1} + \text{Taxa de falta}_{L1} \times \text{Penalidade por falta}_{L1}$$

e

$$\text{Penalidade por falta}_{L1} = \text{Tempo de acerto}_{L2} + \text{Taxa de falta}_{L2} \times \text{Penalidade de falta}_{L2}$$

portanto,

$$\begin{aligned}\text{Tempo médio de acesso à memória} = {} & \text{Tempo de acerto}_{L1} + \text{Taxa de falta}_{L1} \\ & \times (\text{Tempo de acerto}_{L2} + \text{Taxa de falta}_{L2} \times \text{Penalidade por falta}_{L2})\end{aligned}$$

Nessa fórmula, a taxa de falta de segundo nível é medida sobre os remanescentes da cache de primeiro nível. Para evitar ambiguidade, esses termos são adotados aqui para um sistema de cache de dois níveis:

- *Taxa de falta local.* Essa taxa é simplesmente o número de faltas em uma cache dividido pelo número total de acessos à memória para essa cache. Como você poderia esperar, para a cache de primeiro nível isso é igual à Taxa de falta_{L1}, e para a cache de segundo nível é igual à Taxa de falta_{L2}.
- *Taxa de falta global.* O número de faltas na cache dividido pelo número total de acessos à memória gerados pelo processador. Usando os termos anteriores, a taxa de falta global para a cache de primeiro nível ainda é apenas Taxa de falta_{L1}, mas para a cache de segundo nível é Taxa de $\text{falta}_{L1} \times$ Taxa de falta_{L2}.

Essa taxa de falta local é grande para as caches de segundo nível porque a cache de primeiro nível recebe o máximo dos acessos à memória. É por isso que a taxa de falta global é a medida mais útil: ela indica qual fração dos acessos à memória que deixam o processador vai até a memória.

Aqui está um lugar onde brilha a medida das faltas por instrução. Em vez da confusão sobre taxas de falta locais e globais, simplesmente expandimos os stalls da memória por instrução para acrescentar o impacto de uma cache de segundo nível.

$$\begin{aligned}\text{Média de stalls de memória por instrução} &= \text{Faltas por instrução}_{L1} \times \text{Tempo de acerto}_{L2} \\ &+ \text{Faltas por instrução}_{L2} \times \text{Penalidade de falta}_{L2}\end{aligned}$$

Exemplo Suponha que, em 1.000 referências à memória, existam 40 faltas na cache de primeiro nível e 20 faltas na cache de segundo nível. Quais são as diversas taxas de falta? Considere que a penalidade por falta da cache L2 para a memória seja de 200 ciclos de clock, o tempo de acerto da cache L2 seja de 10 ciclos de clock, o tempo de acerto de L1 seja de um ciclo de clock e exista 1,5 referência de memória por instrução. Qual é o tempo médio de acesso à memória e a média de ciclos de stall por instrução? Ignore o impacto das escritas.

Resposta A taxa de falta (seja local ou global) para a cache de primeiro nível é 40/1.000 ou 4%. A taxa de falta local para a cache de segundo nível é 20/40 ou 50%. A taxa de falta global da cache de segundo nível é 20/1.000 ou 2%. Então,

$$\begin{aligned}\text{Tempo médio de acesso à memória} &= \text{Tempo de acerto}_{L1} + \text{Taxa de falta}_{L1} \times (\text{Tempo de acerto}_{L2} \\ &+ \text{Taxa de falta}_{L2} \times \text{Penalidade de falta}_{L2}) \\ &= 1 + 4\% \times (10 + 50\% \times 200) = 1 + 4\% \times 110 = 5,4 \text{ ciclos de clock}\end{aligned}$$

Para ver quantas faltas obtemos por instrução, dividimos 1.000 referências à memória por 1,5 referência à memória por instrução, o que gera 667 instruções. Assim, precisamos multiplicar as faltas por 1,5 para obter o número de faltas por 1.000 instruções. Temos 40 × 1,5 ou 60 faltas L1 e 20 × 1,5, ou 30 faltas L2 por 1.000 instruções. Para a média dos stalls de memória por instrução, considerando que as faltas são distribuídas uniformemente entre instruções e dados:

$$\begin{aligned}\text{Média de stalls de memória por instrução} &= \text{Faltas por instrução}_{L1} \times \text{Tempo de acerto}_{L2} \\ &+ \text{Faltas por instrução}_{L2} \times \text{Penalidade de falta}_{L2} \\ &= (60 / 1.000) \times 10 + (30 / 1.000) \times 200 \\ &= 0,060 \times 10 + 0,030 \times 200 = 6,6 \text{ ciclos de clock}\end{aligned}$$

Se subtrairmos o tempo de acerto L1 do tempo médio de acesso à memória e depois multiplicarmos pelo número médio de referências à memória por instrução, obteremos a mesma média de stalls de memória por instrução:

$$(5,4 - 1,0) \times 1,5 = 4,4 \times 1,5 = 6,6 \text{ ciclos de clock}$$

Conforme mostra esse exemplo, pode haver menos confusão com caches multiníveis quando se calcula usando faltas por instrução, ao contrário das taxas de falta.

Observe que essas fórmulas são para leituras e escritas combinadas, considerando uma cache write-back de primeiro nível. Obviamente, uma cache write-through de primeiro nível enviará *todas* as escritas para o segundo nível, e não apenas as faltas, e um buffer de escrita poderá ser usado.

As Figuras B.14 e B.15 mostram como as taxas de falta e o tempo de execução relativo mudam com o tamanho de uma cache de segundo nível para um projeto. A partir dessas figuras, podemos descobrir duas coisas. A primeira é que a taxa de falta de cache global

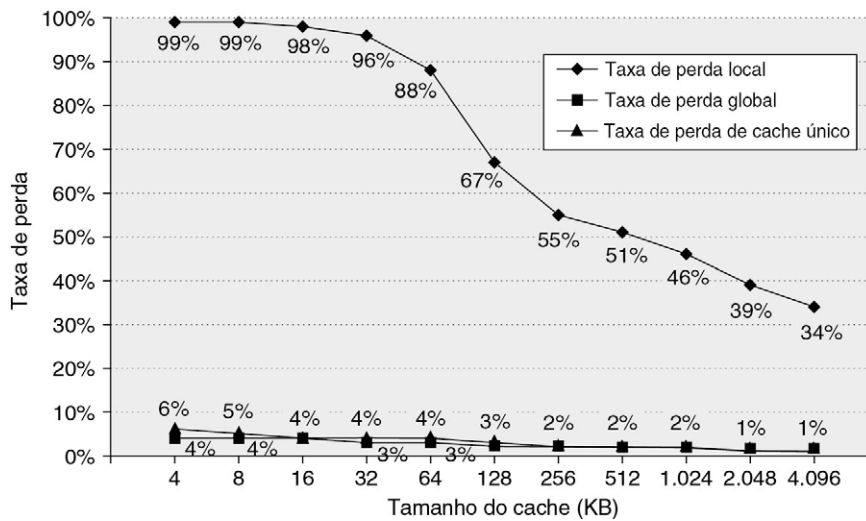



FIGURA B.14 **Taxas de falta *versus* tamanho de cache para caches multiníveis.**
Caches de segundo nível menores que a soma das duas caches de primeiro nível de 64 KB fazem pouco sentido, conforme refletido nas taxas de falta altas. Depois de 256 KB, a única cache está dentro de 10% das taxas de falta globais. A taxa de falta de uma cache de único nível *versus* o tamanho é desenhado contra a taxa de falta local e a taxa de falta global de uma cache de segundo nível usando uma cache de primeiro nível de 32 KB. As caches L2 (unificadas) foram associativas por conjunto com duas vias, com substituição LRU. O tamanho de bloco para caches L1 e L2 foi de 64 bytes. Os dados foram coletados como na Figura B.4.

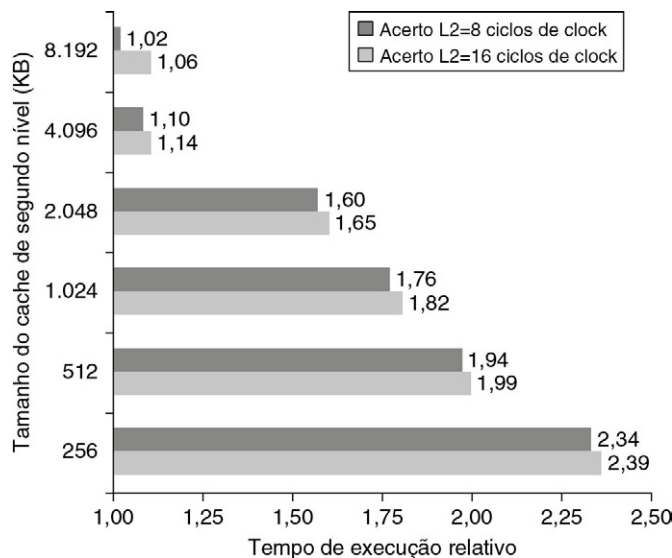



FIGURA B.15 **Tempo de execução relativo pelo tamanho da cache de segundo nível.**
As duas barras são para diferentes ciclos de clock para um acerto na cache L2. O tempo de execução de referência de 1,00 é para uma cache de segundo nível com 8.192 KB, com uma latência de um ciclo de clock em um acerto no segundo nível. Esses dados foram coletados da mesma maneira que na Figura B.14, usando um simulador para imitar o Alpha 21264.

é muito semelhante à taxa de falta de única cache da cache de segundo nível, desde que a cache de segundo nível seja muito maior que a cache de primeiro nível. Daí se aplicam nossa intuição e nosso conhecimento sobre as caches de primeiro nível. A segunda coisa é que a taxa de falta da cache local *não* é uma boa medida das caches secundárias; essa é uma função da taxa de falta da cache de primeiro nível, e por isso pode variar alterando-se a cache de primeiro nível. Assim, a taxa de falta da cache global deverá ser usada na avaliação das caches de segundo nível.

Com essas definições estabelecidas, podemos considerar os parâmetros das caches de segundo nível. A diferença principal entre os dois níveis é que a velocidade da cache de primeiro nível afeta a taxa de clock do processador, enquanto a velocidade da cache de segundo nível só afeta a penalidade por falta da cache de primeiro nível. Assim, podemos considerar muitas alternativas na cache de segundo nível, que poderiam ser mal escolhidas para a cache de primeiro nível. Existem duas perguntas principais no projeto da cache de segundo nível: 1) ela reduzirá a parte do tempo médio de acesso à memória do CPI; 2) quanto isso custa?

A decisão inicial é o tamanho de uma cache de segundo nível. Como tudo na cache de primeiro nível provavelmente estará na cache de segundo nível, este deverá ser muito maior que o primeiro. Se as caches de segundo nível forem apenas um pouco maiores, a taxa de falta local será alta. Essa observação inspira o projeto de enormes caches de segundo nível — do tamanho da memória principal nos computadores mais antigos!

Uma questão é se a associatividade em conjunto faz mais sentido para as caches de segundo nível.

Exemplo Com os dados a seguir, qual é o impacto da associatividade da cache de segundo nível sobre sua penalidade por falta?

- Tempo de acerto$_{L2}$ para mapeamento direto = 10 ciclos de clock.
- Associatividade por conjunto com duas vias aumenta o tempo de acerto por 0,1 ciclo de clock até 10,1 ciclos de clock.
- Taxa de falta local$_{L2}$ para mapeamento direto = 25%.
- Taxa de falta local$_{L2}$ para associatividade por conjunto com duas vias = 20%.
- Penalidade de falta$_{L2}$ = 200 ciclos de clock.

Resposta Para uma cache de segundo nível mapeada diretamente, a penalidade por falta da cache de primeiro nível é

$$\text{Penalidade por falta}_{\text{1 via L2}} = 10 + 25\% \times 200 = 60{,}0 \text{ ciclos de clock}$$

O acréscimo do custo de associatividade aumenta o custo de acerto apenas em 0,1 ciclo de clock, tornando a nova penalidade por falta da cache de primeiro nível

$$\text{Penalidade de falta}_{\text{2 vias L2}} = 10{,}1 + 20\% \times 200 = 50{,}1 \text{ ciclos de clock}$$

Na realidade, as caches de segundo nível são quase sempre sincronizadas com a cache de primeiro nível e o processador. De acordo com isso, o tempo de acerto no segundo nível precisa ser um número inteiro de ciclos de clock. Se tivermos sorte, mantemos o tempo de acerto no segundo nível em 10 ciclos; senão, arredondamos para 11 ciclos. Qualquer escolha é uma melhoria em relação à cache de segundo nível mapeada diretamente:

$$\text{Penalidade de falta}_{\text{2 vias L2}} = 10 + 20\% \times 200 = 50{,}0 \text{ ciclos de clock}$$
$$\text{Penalidade de falta}_{\text{2 vias L2}} = 11 + 20\% \times 200 = 51{,}0 \text{ ciclos de clock}$$

Agora, podemos reduzir a penalidade por falta reduzindo a *taxa de falta* das caches de segundo nível.

Outra consideração refere-se ao fato de os dados na cache de primeiro nível estarem na cache de segundo nível. A *inclusão multinível* é a política natural para hierarquias de memória: dados L1 estão sempre presentes em L2. A inclusão é desejável, porque a consistência entre a E/S e as caches (ou entre as caches em um multiprocessador) pode ser determinada simplesmente verificando-se a cache de segundo nível.

Uma desvantagem da inclusão é que as medições podem sugerir blocos menores para a cache menor de primeiro nível e blocos maiores para a cache maior de segundo nível. Por exemplo, o Pentium 4 possui blocos de 64 bytes em suas caches L1 e blocos de 128 bytes em sua cache L2. A inclusão ainda pode ser mantida com mais trabalho em uma falta no segundo nível. A cache de segundo nível precisa invalidar todos os blocos de primeiro nível que são mapeados para o bloco de segundo nível a serem substituídos, ocasionando uma taxa de falta de primeiro nível ligeiramente maior. Para evitar esses problemas, muitos projetistas de cache mantêm o tamanho do bloco igual em todos os níveis de caches.

Porém, e se o projetista só puder usar uma cache L2 que seja ligeiramente maior que a cache L1? Será que uma parte significativa de seu espaço deve ser usada como uma cópia redundante da cache L1? Nesses casos, uma política oposta sensível é a *exclusão multinível*: dados L1 *nunca* são encontrados em uma cache L2. Normalmente, com a exclusão, uma falta na cache em L1 resulta em uma permuta dos blocos entre L1 e L2, em vez de uma substituição de um bloco L1 por um bloco L2. Essa política impede o desperdício de espaço na cache L2. Por exemplo, um chip AMD Opteron obedece à propriedade de exclusão usando duas caches L1 de 64 KB e uma cache L2 de 1 MB.

Conforme ilustram essas questões, embora um novato possa projetar as caches de primeiro e segundo nível independentes uma da outra, o projetista da cache de primeiro nível tem uma tarefa mais simples, dado uma cache de segundo nível compatível. É uma aposta menor usar um write-through, por exemplo, se houver uma cache write-back no próximo nível para atuar como uma parada para as escritas repetidas e ele usar a inclusão multinível.

A essência de todos os projetos de cache é balancear acertos rápidos e poucas faltas. Para as caches de segundo nível, existem muito menos acertos do que na cache de primeiro nível, de modo que a ênfase passa para menos faltas. Essa *percepção* leva a caches muito maiores e técnicas para reduzir a taxa de falta, como associatividade mais alta e blocos maiores.

Quinta otimização: dando prioridade a faltas de leitura em relação a escritas para reduzir a penalidade por falta

Essa otimização atende às leituras antes que as escritas tenham sido concluídas. Começamos examinando as complexidades de um buffer de escrita.

Com uma cache write-through, a melhoria mais importante é um buffer de escrita do tamanho apropriado. Porém, os buffers de escrita complicam os acessos à memória, pois poderiam manter o valor atualizado de um local necessário em uma falta na leitura.

Exemplo Examine esta sequência de código:

```
SW R3, 512(R0)   ;M[512]←R3   (índice de cache 0)
LW R1, 1024(R0)  ;R1←M[1024]  (índice de cache 0)
LW R2, 512(R0)   ;R2←M[512]   (índice de cache 0)
```

Considere uma cache write-through, mapeada diretamente, que mapeia 512 e 1.024 no mesmo bloco, e um buffer de escrita de quatro palavras que não é verificado em uma falta na leitura. O valor em R2 será sempre igual ao valor em R3?

Resposta Usando a terminologia do Capítulo 2, esse é um hazard de dados de leitura após escrita na memória. Vamos acompanhar um acesso à cache para ver o hazard. Os dados em R3 são colocados no buffer de escrita após o armazenamento. O carregamento seguinte utiliza o mesmo índice de cache e, portanto, é uma falta. A segunda instrução de carregamento tenta colocar o valor no local 512 no registrador R2; isso também resulta em uma falta. Se o buffer de escrita não tivesse completado a escrita no local 512 da memória, a leitura do local 512 colocaria o valor errado antigo no bloco de cache e depois em R2. Sem precauções apropriadas, R3 não seria igual a R2!

O modo mais simples de sair desse dilema é fazer com que a falta na leitura espere até que o buffer de escrita esteja vazio. A alternativa é verificar o conteúdo do buffer de escrita em uma falta na leitura e, se não houver conflitos e o sistema de memória estiver disponível, deixar que a falta na leitura continue. Quase todos os processadores de desktop e servidor utilizam a segunda técnica, dando prioridade às leituras em relação às escritas.

O custo das escritas pelo processador em uma cache write-back também pode ser reduzido. Suponha que uma falta na leitura substituirá um bloco de memória sujo. Em vez de escrever o bloco modificado na memória e depois ler a memória, poderíamos copiar o bloco modificado para um buffer, ler a memória e *depois* escrever na memória. Desse modo, a leitura do processador, pela qual ele está provavelmente esperando, terminará mais cedo. De modo semelhante à situação anterior, se houver uma falta na leitura, o processador poderá ficar em stall até que o buffer esteja vazio ou verificar os endereços das palavras no buffer em busca de conflitos.

Agora que temos cinco otimizações que reduzem as penalidades por falta na cache ou taxas de falta, é hora de vermos a redução do último componente do tempo médio de acesso à memória. O tempo de acerto é crítico porque pode afetar a taxa de clock do processador; em muitos processadores de hoje, o tempo de acesso à cache limita a taxa de ciclo de clock, mesmo para processadores que usam múltiplos ciclos de clock para acessar a cache. Daí um tempo de acerto rápido ser multiplicado em importância, além da fórmula do tempo médio de acesso à memória, pois isso ajuda tudo.

Sexta otimização: evitando tradução de endereço durante a indexação da cache para reduzir o tempo de acerto

Até mesmo uma cache pequena e simples precisa estar de acordo com a tradução de um endereço virtual do processador para um endereço físico, a fim de acessar a memória. Conforme descrevemos na Seção B.4, os processadores tratam a memória principal como apenas outro nível da hierarquia de memória e, por isso, o endereço da memória virtual que existe no disco precisa ser mapeado para a memória principal.

A orientação de tornar o caso comum rápido sugere que usemos endereços virtuais para a cache, pois os acertos são muito mais comuns que as faltas. Essas são as chamadas caches *virtuais*, com a cache *física* usada para identificar a cache tradicional, que usa endereços físicos. Como veremos em breve, é importante distinguir duas tarefas: indexar a cache e

comparar endereços. Assim, a questão é se um endereço virtual ou físico será usado para indexar a cache e se um endereço virtual ou físico será usado na comparação de tags. O endereçamento virtual completo para índices e tags elimina o tempo de tradução de endereço de um acerto na cache. Então, por que nem todos montam caches endereçadas virtualmente?

Um motivo é a proteção. A proteção em nível de página é verificada como parte da tradução de endereço virtual para físico e precisa ser imposta, não importa o que aconteça. Uma solução é copiar a informação de proteção do TLB em uma falta, acrescentar um campo para mantê-la e verificá-la em cada acesso à cache endereçada virtualmente.

Outro motivo é que, toda vez que um processo é trocado, os endereços virtuais se referem a diferentes endereços físicos, exigindo que a cache seja esvaziada. A Figura B.16 mostra o impacto das taxas de falta desse esvaziamento. Uma solução é aumentar a largura da tag de endereço de cache com uma *tag identificadora de processo* (Process-Identifier Tag — PID). Se o sistema operacional atribuir essas tags aos processos, ele só precisará esvaziar a cache quando um PID for reciclado, ou seja, o PID distingue se os dados na cache são ou não para esse programa. A Figura B.16 mostra a melhoria das taxas de falta usando PIDs para evitar esvaziamentos de cache.

Um terceiro motivo pelo qual as caches virtuais não são mais populares é que os sistemas operacionais e os programas do usuário podem utilizar dois endereços virtuais diferentes

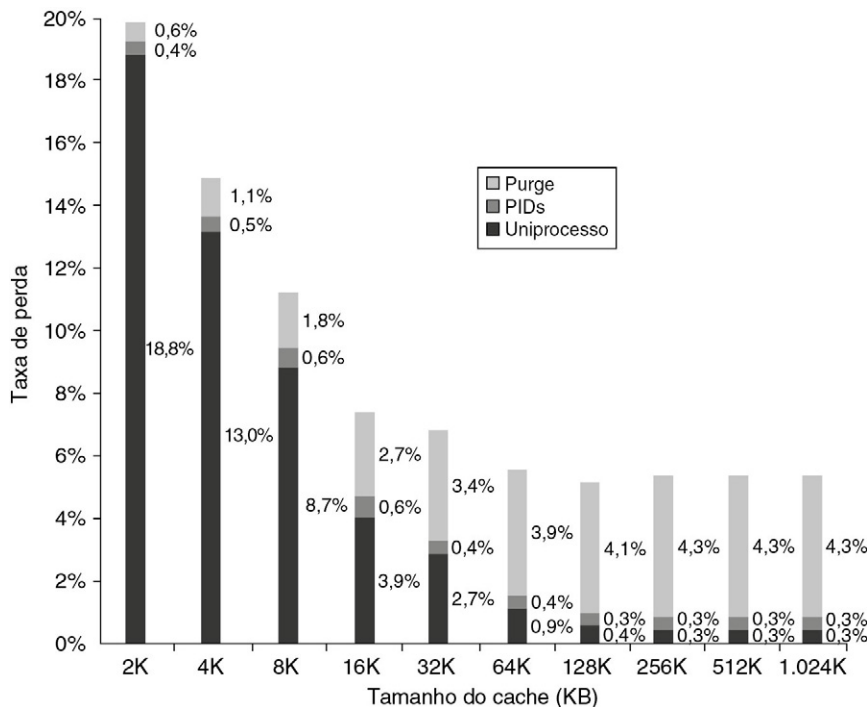



FIGURA B.16 Taxa de falta contra tamanho de cache endereçada virtualmente de um programa medida de três maneiras: sem trocas de processo (uniprocesso), com trocas de processo usando uma tag identificadora de processo (PID) e com trocas de processo mas sem PIDs (purge).
As PIDS aumentam a taxa de falta absoluta uniprocesso em 0,3-0,6% e economizam de 0,6-4,3% em relação ao purging. Agarwal (1987) coletou essas estatísticas para o sistema operacional Ultrix rodando em um VAX, considerando caches mapeadas diretamente com tamanho de bloco de 16 bytes. Observe que a taxa de falta sobe de 128 K para 256 K. Esse comportamento não intuitivo pode ocorrer nas caches porque a mudança do tamanho muda o mapeamento dos blocos de memória para blocos de cache, o que pode mudar a taxa de falta por conflito.

para o mesmo endereço físico. Esses endereços duplicados, chamados *sinônimos* ou *aliases*, poderiam resultar em duas cópias dos mesmos dados em uma cache virtual; se um for modificado, o outro terá o valor errado. Com uma cache física, isso não aconteceria, pois os acessos seriam primeiro traduzidos para o mesmo bloco de cache física.

As soluções de hardware para o problema do sinônimo, chamadas *antialiasing*, garantem a cada bloco de cache um endereço físico exclusivo. O Opteron utiliza uma cache de instruções de 64 KB com uma página de 4 KB e associatividade por conjunto com duas vias, daí o hardware ter de tratar de aliases envolvidos com os três bits de endereço virtuais no índice do conjunto. Isso evita aliases simplesmente verificando todos os oito locais possíveis em uma falta — dois blocos em cada um dos quatro conjuntos — para ter certeza de que nenhum combina com o endereço físico dos dados sendo lidos. Se algum for encontrado, ele será invalidado, de modo que, quando os novos dados forem carregados na cache, seu endereço físico será garantidamente exclusivo.

O software pode tornar esse problema muito mais fácil, forçando os aliases a compartilhar alguns bits de endereço. Uma versão mais antiga do UNIX da Sun Microsystems, por exemplo, exigia que todos os aliases fossem idênticos nos últimos 18 bits de seus endereços; essa restrição é chamada *coloração de página*. Observe que a coloração de página é simplesmente um mapeamento associativo por conjunto aplicado à memória virtual: as páginas de 4 KB (2^{12}) são mapeadas usando 64 (2^6) conjuntos, para garantir que os endereços físicos e virtuais correspondam nos últimos 18 bits. Essa restrição significa que uma cache mapeada diretamente com 2^{18} (256 K) bytes ou menor nunca poderá ter endereços físicos duplicados para os blocos. Do ponto de vista da cache, a coloração de página efetivamente aumenta o offset de página, pois o software garante que os últimos poucos bits do endereço de página virtual e físico são idênticos.

A última área de preocupação com os endereços virtuais é a E/S. A E/S normalmente usa endereços físicos e, portanto, exige que o mapeamento para endereços virtuais interaja com uma cache virtual. (O impacto da E/S sobre as caches será discutido no Apêndice D.)

Uma alternativa para conseguir o melhor das caches virtual e física é usar parte do offset de página — a parte idêntica nos endereços virtual e físico — para indexar a cache. Ao mesmo tempo que a cache está sendo lida usando esse índice, a parte virtual do endereço é traduzida e a combinação de tag usa endereços físicos.

Essa alternativa permite que a leitura da cache comece imediatamente, e ainda assim a comparação de tags se dá com os endereços físicos. A limitação dessa alternativa *indexada virtualmente, marcada fisicamente* é que uma cache mapeada diretamente não pode ser maior do que o tamanho da página. Por exemplo, na cache de dados da Figura B.5, o índice é de 9 bits e o offset do bloco de cache é de 6 bits. Para usar esse truque, o tamanho da página virtual teria de ser pelo menos $2^{(9+6)}$ bytes ou 32 KB. Senão, uma parte do índice precisaria ser traduzida de endereço virtual para físico. A Figura B.17 mostra a organização das caches, o translation lookaside buffers (TLBs) e a memória virtual quando essa técnica é usada.

A associatividade pode manter o índice na parte física do endereço e, ainda assim, admitir uma cache grande. Lembre-se de que o tamanho do índice é controlado por esta fórmula:

$$2^{\text{Índice}} = \frac{\text{Tamanho da cache}}{\text{Tamanho de bloco} \times \text{Associatividade por conjunto}}$$

Por exemplo, dobrar a associatividade e dobrar o tamanho da cache não mudam o tamanho do índice. A cache do IBM 3033, como exemplo extremo, é associativo por conjunto com 16 vias, embora os estudos mostrem que existe pouco benefício nas taxas de falta

acima da associatividade em conjunto com oito vias. Essa alta associatividade permite que uma cache de 64 KB seja endereçada com um índice físico, apesar das páginas de 4 KB na arquitetura da IBM.

Resumo da otimização básica da cache

As técnicas apresentadas nesta seção para melhorar a taxa de falta, a penalidade por falta e o tempo de acerto geralmente têm impacto em outros componentes da equação do tempo médio de acesso à memória, além da complexidade da hierarquia da memória. A Figura B.18 resume essas técnicas e estima o impacto sobre a complexidade, com + significando que a técnica melhora o fator, – significando que prejudica o fator e espaço significando que não tem impacto. Nenhuma otimização nessa figura ajuda mais do que uma categoria.

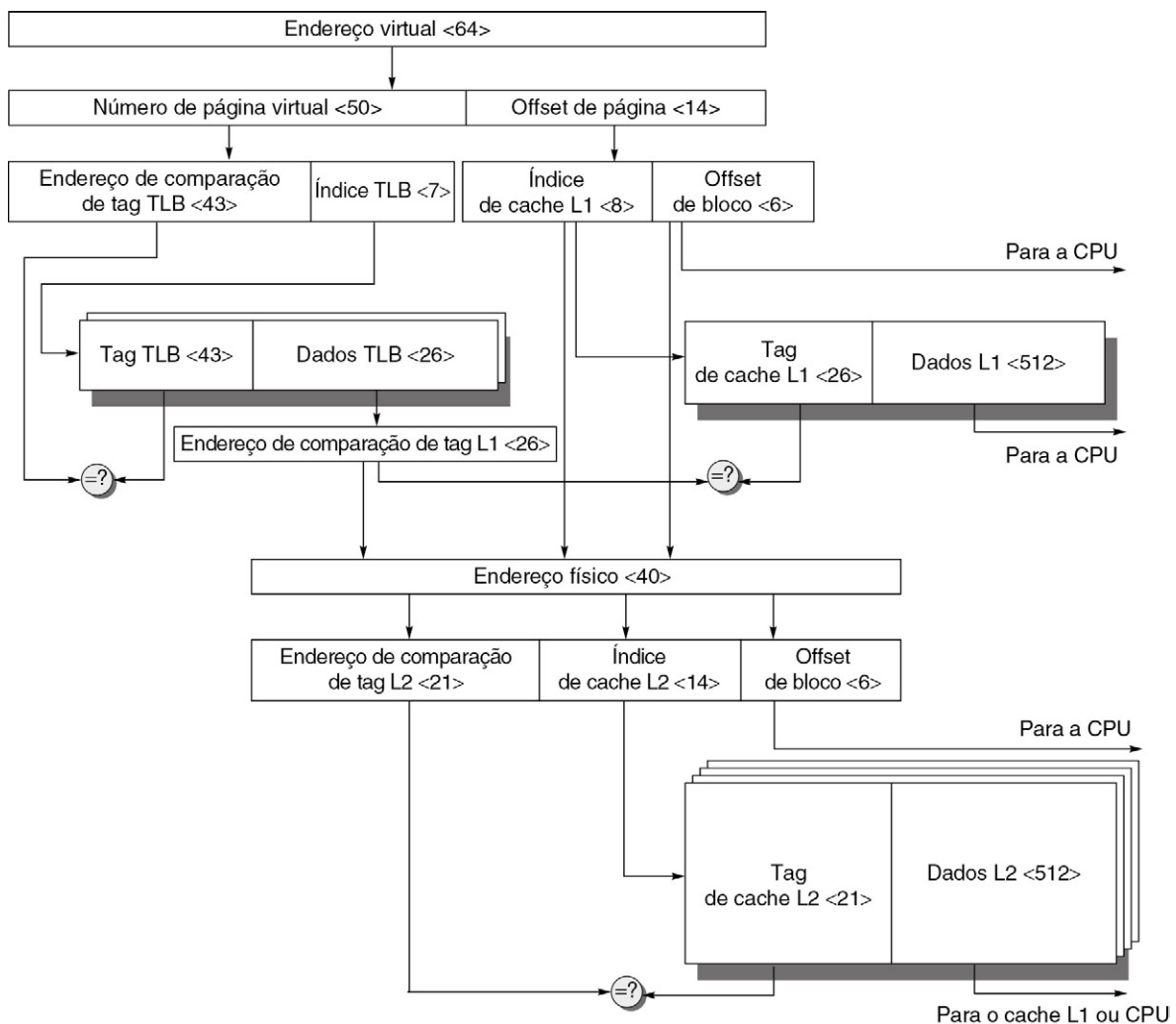



FIGURA B.17 A imagem geral de uma hierarquia de memória hipotética, indo dos endereços virtuais para o acesso à cache L2.
O tamanho da página é de 16 KB. O TLB é associativo por conjunto com duas vias e com 256 entradas. A cache L1 é de 16 KB mapeada diretamente, e a cache L2 é associativa por conjunto de quatro vias com um total de 4 MB. Os dois usam blocos de 64 bytes. O endereço virtual é de 64 bits e o endereço físico é de 40 bits.

B.4 MEMÓRIA VIRTUAL

[…] um sistema foi idealizado para fazer com que a combinação da memória de núcleo apareça ao programador como um armazenamento de único nível, com as transferências de requisitos ocorrendo automaticamente.

Kilburn *et al.* (1962)

A qualquer momento, os computadores estão executando múltiplos processos, cada qual com seu próprio espaço de endereços (os processos serão descritos na próxima seção). Seria muito dispendioso dedicar todo um espaço de endereços de memória a cada processo, especialmente porque muitos processos usam apenas pequena parte de seu espaço de endereços. Logo, é preciso haver um meio de compartilhar uma quantidade menor de memória física entre muitos processos.

Um modo de fazer isso, a *memória virtual*, divide a memória física em blocos e os aloca a diferentes processos. Inerente a tal técnica deve existir um esquema de *proteção* que restrinja um processo aos blocos que pertencem apenas a esse processo. A maioria das formas de memória virtual também reduz o tempo para iniciar um programa, pois nem todo código e dados precisam estar na memória física antes que um programa possa iniciar.

Embora a proteção fornecida pela memória virtual seja essencial para os computadores atuais, o compartilhamento não é o motivo pelo qual a memória virtual foi inventada. Se um programa ficasse muito grande para a memória física, era função do programador fazer com que ele coubesse. Os programadores dividiam os programas em partes, depois identificavam as partes que eram mutuamente exclusivas e carregavam e descarregavam esses *overlays* sob controle do programa do usuário durante a execução. O programador garantia que o programa nunca tentaria acessar mais memória física principal do que existia no computador e que o overlay apropriado era carregado no momento certo. Como você pode imaginar, essa responsabilidade afetava a produtividade do programador.

A memória virtual foi inventada para tirar esse peso das costas dos programadores; ela controla automaticamente os dois níveis da hierarquia de memória representados pela memória principal e armazenamento secundário. A Figura B.19 mostra o mapeamento entre a memória virtual e a memória física para um programa com quatro páginas.

Técnica	Tempo de acerto	Penalidade de falta	Taxa de falta	Complexidade do hardware	Comentário
Tamanho de bloco maior		–	+	0	Trivial; L2 do Pentium 4 usa 128 bytes
Tamanho de cache maior	–		+	1	Muito usado, especialmente para caches L2
Associatividade mais alta	–		+	1	Muito usado
Caches multiníveis		+		2	Hardware caro; mais difícil se tamanho do bloco L1 ≠ tamanho do bloco L2; muito usado
Prioridade de leitura sobre a escrita		+		1	Muito usado
Evitar tradução de endereço durante indexação da cache	+			1	Muito usado

FIGURA B.18 **Resumo das otimizações básicas de cache mostrando o impacto sobre o desempenho da cache e a complexidade para as técnicas neste apêndice.**

Geralmente, uma técnica ajuda apenas um fator. + significa que a técnica melhora o fator, – significa que ela prejudica esse fator, e um espaço significa que ela não tem impacto. A medida de complexidade é subjetiva, com 0 sendo o mais fácil e 3 sendo um desafio.

Além de compartilhar o espaço de memória protegida e gerenciar automaticamente a hierarquia da memória, a memória virtual também simplifica o carregamento do programa para execução. Chamado *relocação*, esse mecanismo permite que o mesmo programa seja executado em qualquer local na memória física. O programa na Figura B.19 pode ser colocado em qualquer lugar na memória física ou disco simplesmente alterando o mapeamento entre eles (antes da popularidade da memória virtual, os processadores incluíam um registrador de relocação apenas para essa finalidade). Uma alternativa a uma solução de hardware seria um software que mudasse os endereços em um programa toda vez que fosse executado.

Várias ideias de hierarquia geral da memória a respeito de caches, do Capítulo 1, são semelhantes à memória virtual, embora muitos dos termos sejam diferentes. *Página* ou *segmento* é usado para o bloco, e *falta de página* (*page fault*) ou *falta de endereço* (*address fault*) é usado para a falta (*miss*). Com a memória virtual, o processador produz *endereços virtuais* que são traduzidos por uma combinação de hardware e software para *endereços físicos*, que acessam a memória principal. Esse processo é chamado de *mapeamento de memória* ou *tradução de endereço*. Hoje, os dois níveis de hierarquia de memória controlados pela memória virtual são DRAMs e discos magnéticos. A Figura B.20 mostra um intervalo típico de parâmetros de hierarquia de memória para a memória virtual.

Existem outras diferenças entre as caches e a memória virtual, além das quantitativas, mencionadas na Figura B.20:

- A substituição nas faltas de cache é controlada principalmente pelo hardware, enquanto a substituição da memória virtual é controlada principalmente pelo sistema operacional. A penalidade por falta maior significa que é mais importante tomar uma boa decisão, de modo que o sistema operacional possa estar envolvido e gastar tempo decidindo o que substituir.
- O tamanho de endereço do processador determina o tamanho da memória virtual, mas o tamanho da cache independe do tamanho de endereço do processador.

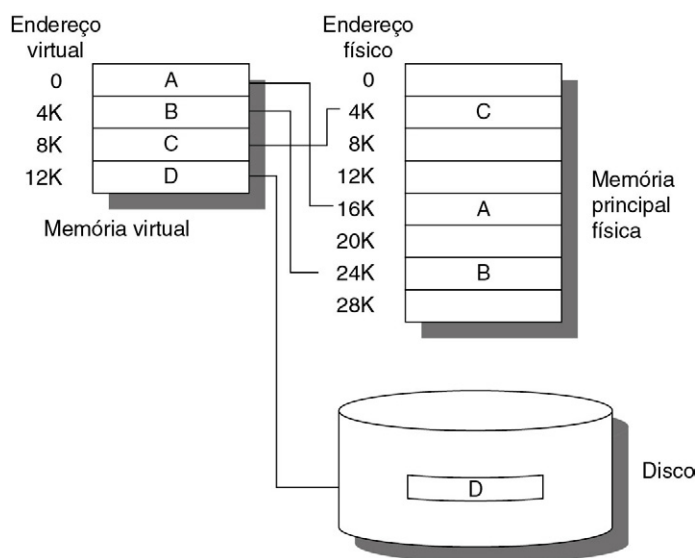



FIGURA B.19 **O programa lógico em seu espaço de endereço virtual contíguo aparece à esquerda.** Ele consiste em quatro páginas: A, B, C e D. O local real de três dos blocos está na memória física principal, e o outro está localizado no disco.

Parâmetro	Cache de primeiro nível	Memória virtual
Tamanho do bloco (página)	16-128 bytes	4.096-65.536 bytes
Tempo de acerto	1-3 ciclos de clock	100-200 ciclos de clock
Penalidade por falta	8-200 ciclos de clock	1.000.000-10.000.000 ciclos de clock
(tempo de acesso)	(6-160 ciclos de clock)	(800.000-8.000.000 ciclos de clock)
(tempo de transferência)	(2-40 ciclos de clock)	(200.000-2.000.000 ciclos de clock)
Taxa de falta	0,1-10%	0,00001-0,001%
Mapeamento de endereço	25-45 bits de endereço físico para 14-20 bits de endereço de cache	32-64 bits de endereço virtual para 25-45 bits de endereço físico

FIGURA B.20 Intervalos típicos de parâmetros para caches e memória virtual.
Os parâmetros da memória virtual representam aumentos de 10-1.000.000 de vezes em relação aos parâmetros da cache. Normalmente, as caches de primeiro nível contêm no máximo 1 MB de dados, enquanto a memória física contém de 256 MB a 1 TB.

- Além de atuar como um armazenamento de apoio de nível inferior para a memória principal na hierarquia, o armazenamento secundário também é usado para o sistema de arquivos. Na verdade, o sistema de arquivos ocupa a maior parte do armazenamento secundário. Normalmente, ele não está no espaço de endereços.

A memória virtual também abrange várias técnicas relacionadas. Os sistemas de memória virtual podem ser categorizados em duas classes: aqueles com blocos de tamanho fixo, chamados *páginas*, e aqueles com blocos de tamanho variável, chamados *segmentos*. As páginas normalmente são fixadas em 4.096-8.192 bytes, enquanto o tamanho do segmento varia. O maior segmento admitido em qualquer processador varia de 2^{16} bytes a 2^{32} bytes; o menor segmento é de 1 byte. A Figura B.21 mostra como as duas técnicas poderiam dividir código e dados.

A decisão de usar a memória virtual paginada *versus* a memória virtual segmentada afeta o processador. O endereçamento paginado tem um único endereço de tamanho fixo, dividido em número de página e offset dentro de uma página, semelhante ao endereçamento da cache. Um único endereço não funciona para endereços segmentados; o tamanho variável dos segmentos exige uma palavra para um número de segmento e uma palavra para um offset dentro de um segmento, um total de duas palavras. Um espaço de endereços não segmentado é mais simples para o compilador.

Os prós e os contras dessas duas técnicas foram bem documentados nos livros sobre sistemas operacionais; a Figura B.22 resume os argumentos. Hoje, devido ao problema da substituição (a terceira linha da figura), poucos computadores utilizam a segmentação pura. Alguns usam uma técnica híbrida, chamada *segmentos paginados*, em que um segmento é um número inteiro de páginas. Isso simplifica a substituição, pois a memória não precisa ser contígua, e os segmentos inteiros não precisam estar na memória principal.

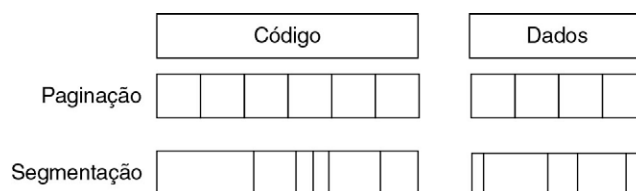



FIGURA B.21 Exemplo de como a paginação e a segmentação dividem um programa.

	Página	Segmento
Palavras por endereço	Uma	Duas (segmento e offset)
Visível ao programador?	Invisível ao programador de aplicações	Pode ser visível ao programador de aplicações
Substituindo um bloco	Trivial (todos os blocos têm o mesmo tamanho)	Difícil (precisa encontrar uma parte contígua, de tamanho variável e não usada da memória principal)
Ineficiência de uso da memória	Fragmentação interna (parte não usada da página)	Fragmentação externa (partes não usadas da memória principal)
Tráfego de disco eficiente	Sim (ajuste do tamanho da página para balancear o tempo de acesso e o tempo de transferência)	Nem sempre (pequenos segmentos podem transferir apenas alguns bytes)

FIGURA B.22 **Paginação contra segmentação.**

Ambas podem desperdiçar memória, dependendo do tamanho do bloco e de como os segmentos se encaixam na memória principal. As linguagens de programação com ponteiros irrestritos exigem que o segmento e o endereço sejam passados. Uma técnica híbrida, chamada *segmentos paginados*, alcança o melhor dos dois mundos: segmentos são compostos de páginas, de modo que a substituição de um bloco é fácil, enquanto um segmento pode ser tratado como uma unidade lógica.

Um híbrido mais recente é um computador oferecendo múltiplos tamanhos de página, com os tamanhos maiores sendo potências de duas vezes o menor tamanho de página. O processador embarcado 405CR da IBM, por exemplo, permite 1 KB, 4 KB ($2^2 \times 1$ KB), 16 KB ($2^4 \times 1$ KB), 64 KB ($2^6 \times 1$ KB), 256 KB ($2^8 \times 1$ KB), 1.024 KB ($2^{10} \times 1$ KB) e 4.096 KB ($2^{12} \times 1$ KB) para atuar como uma única página.

Revisão das quatro perguntas sobre hierarquia da memória

Agora, estamos prontos para responder às quatro perguntas sobre hierarquia de memória para a memória virtual.

P1: Em que parte da memória principal um bloco pode ser alocado?

A penalidade por falta para a memória virtual envolve o acesso a um dispositivo de armazenamento magnético rotativo e, portanto, é muito alta. Dada a escolha de taxas de falta mais baixas ou um algoritmo de posicionamento mais simples, os projetistas de sistemas operacionais normalmente escolhem taxas de falta mais baixas, devido à penalidade por falta exorbitante. Assim, os sistemas operacionais permitem que os blocos sejam alocados em qualquer lugar na memória principal. De acordo com a terminologia da Figura B.2, essa estratégia seria rotulada como totalmente associativa.

P2: Como um bloco é localizado se estiver na memória principal?

Tanto a paginação quanto a segmentação contam com uma estrutura de dados que é indexada por número de página ou segmento. Essa estrutura de dados contém o endereço físico do bloco. Para a segmentação, o offset é acrescentado ao endereço físico do segmento para obter o endereço físico final. Para a paginação, o offset é simplesmente concatenado com esse endereço de página físico (Fig. B.23).

Essa estrutura de dados, contendo os endereços de páginas físicas, normalmente toma a forma de uma *tabela de página*. Indexado pelo número de página virtual, o tamanho da tabela é o número de páginas no espaço de endereços virtuais. Dado um endereço virtual de 32 bits, páginas de 4 KB e 4 bytes por entrada na tabela de página (Page Table Entry — PTE), o tamanho da tabela de página seria $(2^{32}/2^{12}) \times 2^2 = 2^{22}$ ou 4 MB.

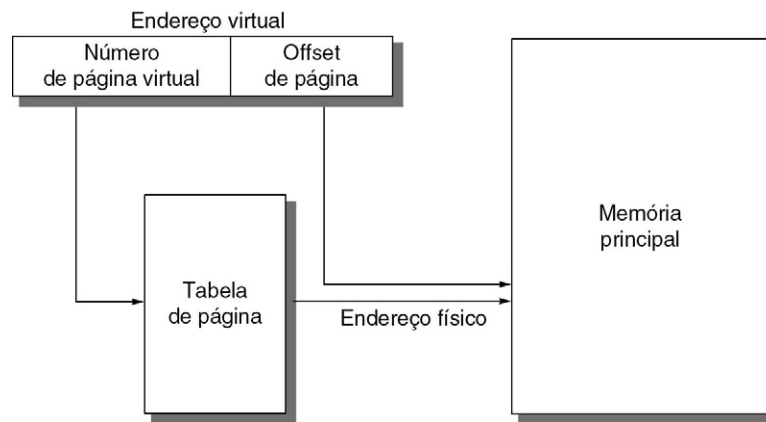



FIGURA B.23 Mapeamento de um endereço virtual para um endereço físico por meio de uma tabela de página.

Para reduzir o tamanho dessa estrutura de dados, alguns computadores aplicam uma função de hashing ao endereço virtual. O hash permite que a estrutura de dados tenha o tamanho do número de páginas *físicas* na memória principal. Esse número poderia ser muito menor do que o número de páginas virtuais. Tal estrutura é chamada de *tabela de página invertida*. Usando o exemplo anterior, uma memória física de 512 MB só precisaria de 1 MB (8×512 MB/4 KB) para uma tabela de página invertida; os 4 bytes extras por entrada da tabela de página são para o endereço virtual. O HP/Intel IA-64 cobre as duas bases, oferecendo tabelas de página tradicionais *e* tabelas de página invertidas, deixando a escolha do mecanismo para o programador do sistema operacional.

Para reduzir o tempo de tradução de endereço, os computadores usam uma cache dedicada a essas traduções de endereço, chamada *translation lookaside buffer* ou simplesmente *translation buffer*, descrito com detalhes em breve.

P3: Qual bloco deverá ser substituído em caso de falta da memória virtual?

Como já dissemos, a diretriz principal do sistema operacional é minimizar as faltas de página. Coerentes com essa diretriz, quase todos os sistemas operacionais tentam substituir o bloco usado menos recentemente (LRU) porque, se o passado prevê o futuro, esse é o menos provavelmente necessário.

Para ajudar o sistema operacional a estimar o LRU, muitos processadores oferecem um *bit de uso* ou *bit de referência*, que é definido logicamente sempre que uma página é acessada. (Para reduzir o trabalho, ele é realmente definido apenas em uma falta do *translation buffer*, que será descrita em breve.) Periodicamente, o sistema operacional apaga os bits de uso e depois os registra de modo que possa determinar quais páginas foram acessadas durante determinado período de tempo. Acompanhando dessa maneira, o sistema operacional pode selecionar uma página que está entre as referenciadas menos recentemente.

P4: O que acontece em uma escrita?

O nível abaixo da memória principal contém discos magnéticos rotativos que exigem milhões de ciclos de clock para acessar. Devido à grande discrepância no tempo de acesso, ninguém criou ainda um sistema operacional de memória virtual que escreva diretamente da memória principal para o disco em cada armazenamento pelo processador. (Esse comentário não deverá ser interpretado como uma oportunidade de se tornar famoso sendo o primeiro a criar um!). Assim, a estratégia de escrita é sempre write-back.

Como o custo de um acesso desnecessário para o próximo nível inferior é muito alto, os sistemas de memória virtual normalmente incluem um bit de modificação. Ele permite que os blocos sejam escritos em disco somente se tiverem sido alterados desde que foram lidos do disco.

Técnicas para uma tradução de endereço rápida

As tabelas de página normalmente são tão grandes que são armazenadas na memória principal e, às vezes, elas mesmas são paginadas. A paginação significa que cada acesso à memória utiliza pelo menos o dobro do tempo, com um primeiro acesso à memória para obter o endereço físico e um segundo acesso para obter os dados. Como dissemos no Capítulo 2, usamos a localidade para evitar o acesso extra à memória. Mantendo as traduções de endereço em uma cache especial, um acesso à memória raramente exige um segundo acesso para traduzir os dados. Essa cache de tradução de endereço especial é conhecida como translation lookaside *buffer* (TLB), também chamado *buffer de tradução* (*translation buffer* — TB).

Uma entrada do TLB é como uma entrada da cache onde a tag mantém partes do endereço virtual e a parte de dados mantém um número de frame da página física, do campo de proteção, do bit de validade e, normalmente, de um bit de uso e um bit de modificação. Para alterar o número do frame da página física ou a proteção de uma entrada na tabela de página, o sistema operacional precisa certificar-se de que a entrada antiga não está no TLB; caso contrário, o sistema não se comportará corretamente. Observe que esse bit de modificação significa que a *página* correspondente foi modificada, e não que a tradução de endereço no TLB está modificado nem que um bloco em particular na cache de dados está modificado. O sistema operacional reinicia esses bits alterando o valor na tabela de página e, depois, invalidando a entrada de TLB correspondente. Quando a entrada é recarregada da tabela de página, o TLB recebe uma cópia precisa dos bits.

A Figura B.24 mostra a organização do TLB de dados do Opteron, com cada etapa da tradução sendo rotulada. Esse TLB usa o mapeamento totalmente associativo; assim, a tradução começa (etapas 1 e 2) enviando o endereço virtual a todas as tags. Naturalmente, a tag precisa ser marcada como válida para permitir uma comparação. Ao mesmo tempo, o tipo de acesso à memória é verificado em busca de uma violação (também na etapa 2) contra informações de proteção no TLB.

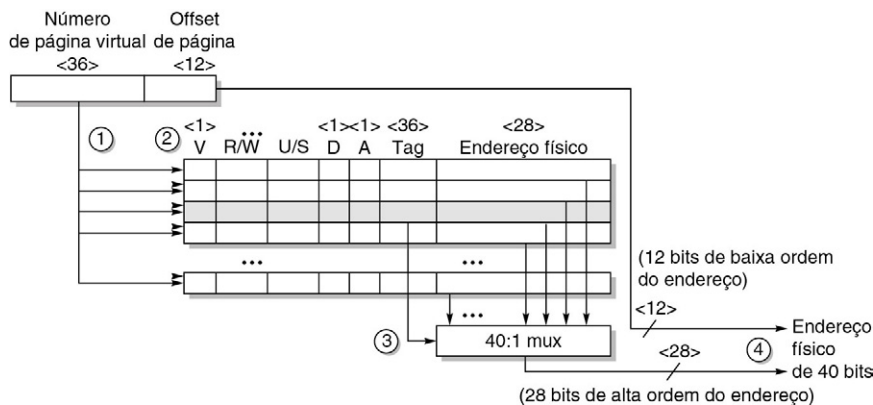



FIGURA B.24 **Operação do TLB de dados do Opteron durante a tradução de endereço.**
As quatro etapas do acerto do TLB aparecem como números circulados. Esse TLB possui 40 entradas. A Seção B.5 descreve os diversos campos de proteção e acesso de uma entrada de tabela de página do Opteron.

Por motivos semelhantes aos do caso da cache, não é preciso incluir os 12 bits do offset de página no TLB. A tabela que combina envia o endereço físico correspondente efetivamente através de um multiplexador 40:1 (etapa 3). O offset de página é, então, combinado com o frame de página física para formar um endereço físico completo (etapa 4). O tamanho do endereço é de 40 bits.

A tradução de endereço pode estar facilmente no caminho crítico determinando o ciclo de clock do processador, de modo que o Opteron utiliza caches L1 endereçadas virtualmente e marcadas fisicamente com tags.

Selecionando um tamanho de página

O parâmetro de arquitetura mais óbvio é o tamanho da página. A escolha da página é uma questão de equilibrar as forças que favorecem um tamanho de página maior contra aquelas que favorecem um tamanho menor. Os seguintes fatores favorecem um tamanho maior:

- O tamanho da tabela de página é inversamente proporcional ao tamanho da página; a memória (ou outros recursos usados para o mapa de memória) pode, portanto, ser economizada, tornando as páginas maiores.
- Como dissemos na Seção B.3, um tamanho de página maior pode permitir caches maiores com menores tempos de acerto na cache.
- A transferência de páginas maiores de/para o armazenamento secundário, possivelmente por uma rede, é mais eficiente do que a transferência de páginas menores.
- O número de entradas do TLB é restrito, de modo que um tamanho de página maior significa que mais memória pode ser mapeada de modo eficiente, reduzindo assim o número de faltas do TLB.

É por esse motivo final que os microprocessadores recentes decidiram dar suporte a múltiplos tamanhos de página; para alguns programas, as faltas de TLB podem ser tão significativas no CPI quanto as faltas de cache.

A motivação principal para um tamanho de página menor é economizar armazenamento. Um tamanho de página pequeno resultará em menos armazenamento desperdiçado quando uma região contígua de memória virtual não for igual em tamanho a um múltiplo do tamanho de página. O nome para essa memória não usada em uma página é *fragmentação interna*. Supondo que cada processo tenha três segmentos principais (texto, heap e pilha), o armazenamento médio desperdiçado por processo será 1,5 vez o tamanho da página. Essa quantidade é insignificante para computadores com centenas de megabytes de memória e tamanhos de página de 4-8 KB. Naturalmente, quando os tamanhos de página se tornam muito grandes (mais de 32 KB), o armazenamento (tanto principal quanto secundário) poderá ser desperdiçado, assim como a largura de banda de E/S. Um aspecto final é o tempo de partida do processo; muitos processos são pequenos, de modo que um tamanho de página grande esticaria o tempo para invocar um processo.

Resumo de memória virtual e caches

Com a memória virtual, os TLBs, as caches de primeiro e de segundo nível — todos mapeando partes do espaço de endereço virtual e físico —, podem ficar confusos sobre quais bits vão para onde. A Figura B.25 oferece um exemplo hipotético, passando de um endereço virtual de 64 bits para um endereço físico de 41 bits, com dois níveis de cache. Essa cache L1 é indexada virtualmente e marcada fisicamente, pois o tamanho da cache e o tamanho da página são de 8 KB. A cache L2 é de 4 MB. O tamanho do bloco para ambos é de 64 bytes.

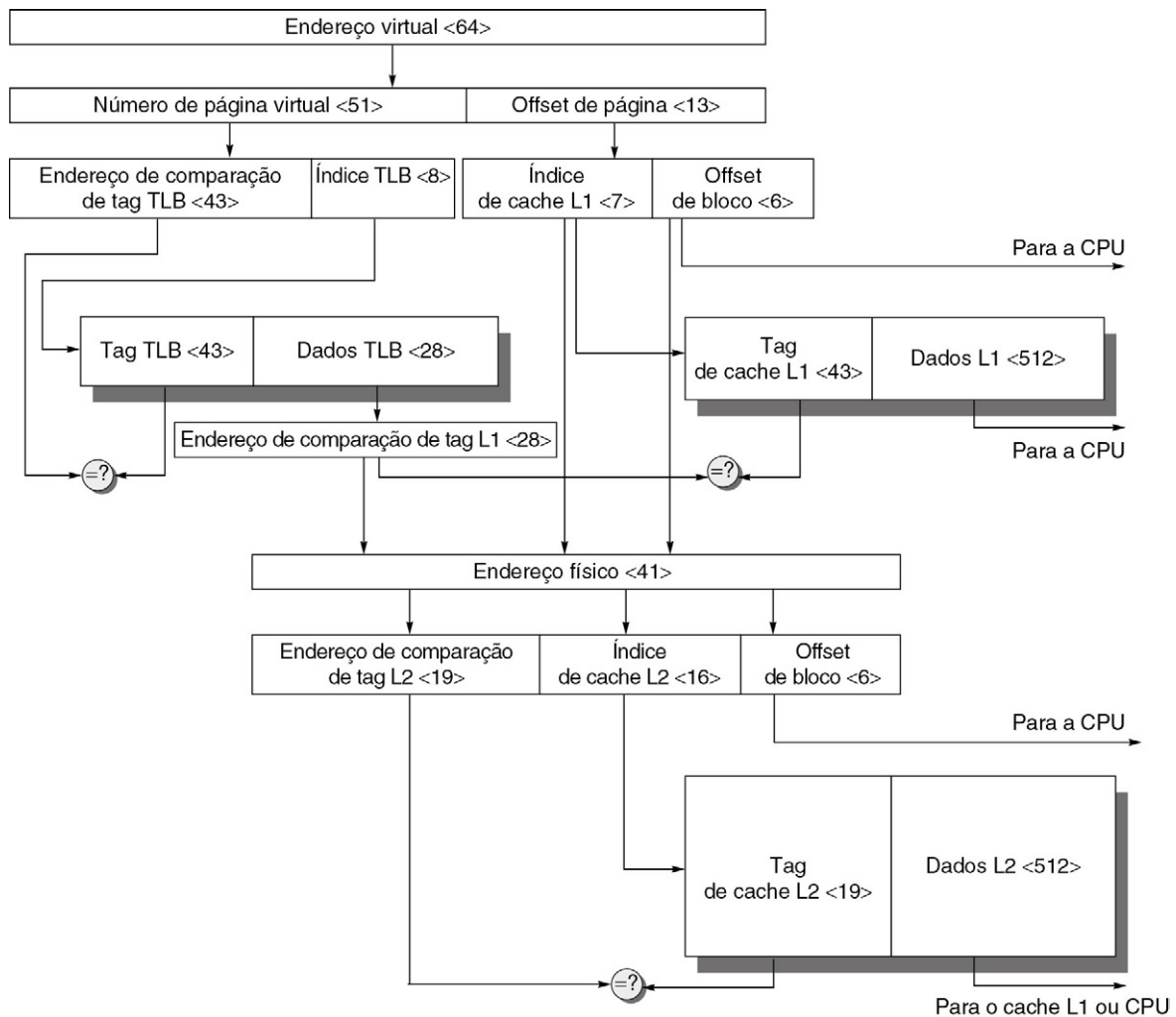



FIGURA B.25 **Imagem geral de uma hierarquia de memória hipotética passando do endereço virtual para o acesso à cache L2.** O tamanho de página é de 8 KB. O TLB é mapeado diretamente com 256 entradas. A cache L1 é mapeada diretamente com 8 KB, e a cache L2 é mapeada diretamente com 4 MB. Ambos usam blocos de 64 bytes. O endereço virtual ocupa 64 bits e o endereço físico ocupa 41 bits. A principal diferença entre esta figura simples e uma cache real é a replicação das partes da figura.

Primeiro, o endereço virtual de 64 bits é dividido logicamente em um número de página virtual e offset de página. O primeiro é enviado ao TLB para ser traduzido para um endereço físico, e os bits mais significativos do segundo são enviados à cache L1 para atuar como um índice. Se a comparação do TLB resultar em igualdade, então o número de página físico é enviado à tag da cache L1 para verificar uma comparação. Se for igual, é um acerto na cache L1. O offset de bloco então seleciona a palavra para o processador.

Se a verificação da cache L1 resultar em uma falta, o endereço físico é então usado para testar a cache L2. A parte do meio do endereço físico é usada como um índice para a cache L2 de 4 MB. A tag da cache L2 resultante é comparada com a parte superior do endereço físico para comparação. Se for igual, temos um acerto na cache L2, e os dados são enviados ao processador, que usa o offset de bloco para selecionar a palavra desejada. Em uma falta na L2, o endereço físico é então usado para obter o bloco da memória.

Embora esse seja um exemplo simples, a principal diferença entre esse desenho e uma cache real é a replicação. Primeiro, há apenas uma cache L1. Quando existem duas caches L1, a metade superior do diagrama é duplicada. Observe que isso levaria a dois TLBs, o que é típico. Daí uma cache e TLB serem para instruções, controladas pelo PC, e uma cache e TLB serem para dados, controlados pelo endereço efetivo.

A segunda simplificação é que todas as caches e TLBs são mapeados diretamente. Se qualquer um fosse associativo por conjunto com *n* vias, então replicaríamos cada conjunto de memória de tag, comparadores e memória de dados *n* vezes e conectaríamos as memórias de dados com um multiplexador *n*:1 para selecionar um acerto. Naturalmente, se o tamanho total da cache permanecesse igual, o índice da cache também encolheria por log2*n* bits, de acordo com a Figura B.7, na página B-19.

B.5 PROTEÇÃO E EXEMPLOS DE MEMÓRIA VIRTUAL

A invenção da multiprogramação, pela qual um computador seria compartilhado por vários programas sendo executados simultaneamente, levou a novas demandas para proteção e compartilhamento entre programas. Essas demandas estão bastante ligadas à memória virtual nos computadores de hoje, e por isso abordamos o assunto aqui, junto com dois exemplos de memória virtual.

A multiprogramação leva ao conceito de *processo*. Metaforicamente, um processo é o ar que o programa respira, o espaço em que ele vive, ou seja, um programa em execução mais qualquer status necessário para continuar executando-o. Tempo compartilhado (*time sharing*) é uma variação da multiprogramação, que compartilha o processador e a memória com vários usuários interativos ao mesmo tempo, dando a ilusão de que todos os usuários têm seus próprios computadores. Assim, a qualquer momento, deverá ser possível passar de um processo para outro. Essa troca é chamada *troca de processo* ou *troca de contexto*.

Um processo precisa operar corretamente, não importando se ele executa continuamente do início ao fim ou se é interrompido repetidamente e trocado com outros processos. A responsabilidade por manter o comportamento correto do processo é compartilhada pelos projetistas do computador e do sistema operacional. O projetista do computador precisa garantir que parte do estado processador, relativo à execução do processo, pode ser salva e restaurada. O projetista do sistema operacional precisa garantir que os processos não interfiram na computação um do outro.

O modo mais seguro de proteger o estado de um processo do outro seria copiar a informação atual em disco. Porém, uma troca de processo levaria segundos — muito tempo para um ambiente de tempo compartilhado.

Esse problema é solucionado pelos sistemas operacionais particionando a memória principal de modo que vários processos diferentes tenham seu estado na memória ao mesmo tempo. Essa divisão significa que o projetista do sistema operacional precisa da ajuda do projetista do computador para oferecer proteção para que um processo não possa modificar outro. Além da proteção, os computadores também providenciam compartilhamento de código e dados entre os processos, para permitir a comunicação entre os processos e economizar memória, reduzindo o número de cópias de informações idênticas.

Protegendo processos

Os processos podem ser protegidos uns dos outros tendo suas próprias tabelas de página, cada qual apontando para páginas de memória distintas. Obviamente, os programas do usuário precisam ser impedidos de modificar suas tabelas de página ou a proteção seria contornada.

A proteção pode ser escalada, dependendo da apreensão do projetista de computador ou do comprador. *Anéis* acrescentados à estrutura de proteção do processador expandem a proteção de acesso à memória a partir de dois níveis (usuário e kernel) para muito mais. Assim como um sistema de classificação militar ultrassecreto, secreto, confidencial e não confidencial, os anéis concêntricos dos níveis de segurança permitem que os mais confiáveis acessem qualquer coisa, o segundo mais confiável tenha acesso a tudo, menos ao nível mais interno, e assim por diante. Os programas "civis" são os menos confiáveis e, portanto, têm intervalo de acessos mais limitado. Também pode haver restrições sobre as partes da memória que podem conter código — proteção de execução — e ainda sobre o ponto de entrada entre os níveis. A estrutura de proteção do Intel 80x86, que utiliza anéis, será descrito mais adiante nesta seção. Não está claro se os anéis são uma melhoria na prática em relação ao sistema simples, que utiliza modos usuário e kernel.

À medida que o projetista passa a ficar apreensivo, esses anéis simples podem não ser suficientes. Restringir a liberdade dada a um programa no átrio interno exige um novo sistema de classificação. Em vez de um modelo militar, a analogia desse sistema é feita com chaves e trancas: um programa não pode destrancar o acesso aos dados, a menos que tenha a chave. Para que essas chaves (ou *capacidades*) sejam úteis, o hardware e o sistema operacional precisam ser capazes de passá-las explicitamente de um programa para outro sem permitir que um programa as falsifique. Essa verificação exige muito suporte do hardware pois o tempo para verificar as chaves precisar ser curto.

A arquitetura do 80x86 experimentou várias dessas alternativas com o passar dos anos. Como a compatibilidade é uma das diretrizes dessa arquitetura, as versões mais recentes da arquitetura incluem todos os seus experimentos na memória virtual. Veremos duas das opções aqui: primeiro, o espaço de endereços segmentados mais antigo; depois, o espaço de endereços plano de 64 bits, mais recente.

Exemplo de memória virtual segmentada: proteção no Intel Pentium

> *O segundo sistema é o sistema mais perigoso que um homem jamais projeta... A tendência geral é exagerar no projeto do segundo sistema, usando todas as ideias e enfeites que foram cuidadosamente deixados de lado no primeiro.*
>
> **F. P. Brooks Jr.**
> *The Mythical Man-Month* (1975)

O 8086 original usava segmentos para o endereçamento, embora não fornecesse nada para a memória virtual ou para a proteção. Os segmentos tinham registradores de base, mas nenhum registrador vinculado e nenhuma verificação de acesso, e, antes que um registrador de segmento pudesse ser carregado, o segmento correspondente tinha que estar na memória física. A dedicação da Intel à memória virtual e à proteção é evidente nos sucessores do 8086, com alguns campos estendidos para dar suporte a endereços maiores. Esse esquema de proteção é elaborado com muitos detalhes projetados cuidadosamente para tentar evitar as brechas de segurança. Vamos nos referir a ele como IA-32. As próximas páginas destacam algumas das medidas de segurança da Intel; se você achar a leitura difícil, imagine a dificuldade para implementá-las!

A primeira melhoria é dobrar o modelo de proteção tradicional de dois níveis: o IA-32 possui quatro níveis de proteção. O nível mais interno (0) corresponde ao modo kernel tradicional, e o nível mais externo (3) é o modo menos privilegiado. O IA-32 possui pilhas separadas para cada nível, para evitar brechas de segurança entre os níveis. Há também estruturas de dados semelhantes às tabelas de página tradicionais, que contêm

os endereços físicos para segmentos, além de uma lista de verificações a serem feitas nos endereços traduzidos.

Os projetistas da Intel não pararam aí. O IA-32 divide o espaço de endereços, permitindo que tanto o sistema operacional quanto o usuário acessem o espaço inteiro. O usuário do IA-32 pode chamar uma rotina do sistema operacional nesse espaço e até mesmo passar parâmetros para ele, enquanto retém proteção total. Essa chamada segura não é uma ação trivial, pois a pilha para o sistema operacional é diferente da pilha do usuário. Além do mais, o IA-32 permite que o sistema operacional mantenha o nível de proteção da rotina *chamada* para os parâmetros que são passados a ela. Essa brecha de proteção em potencial é impedida, não permitindo que o processo do usuário peça que o sistema operacional acesse indiretamente algo que não teria sido capaz de acessar por si só (essas brechas de segurança são chamadas *cavalos de Troia — Trojan horses*).

Os projetistas da Intel foram guiados pelo princípio de confiar no sistema operacional o mínimo possível, enquanto dão suporte ao compartilhamento e à proteção. Como exemplo do uso de tal compartilhamento protegido, suponha que um programa de folha de pagamento escreva cheques e também atualize os valores acumulados no ano sobre os pagamentos totais de salário e benefícios. Assim, queremos dar ao programa a capacidade de ler a informação de salário e o acumulado no ano, e modificar a informação de acumulado no ano, mas não o salário. Em breve, veremos o mecanismo para dar suporte a esses recursos. No restante desta subseção, veremos a figura completa da proteção do IA-32 e examinaremos sua motivação.

Acrescentando verificação de limites e mapeamento de memória

O primeiro passo na melhoria do processador da Intel foi fazer com que o endereçamento segmentado verificasse os limites, além de fornecer uma base. Em vez de um endereço-base, os registradores de segmento no IA-32 contêm um índice para uma estrutura de dados da memória virtual chamada *tabela de descritores*. As tabelas de descritores desempenham o papel das tabelas de página tradicionais. No IA-32, o equivalente de uma entrada de tabela de página é um *descritor de segmento*. Ele contém campos encontrados nos PTEs:

- *Bit de presença.* Equivalente ao bit de validade do PTE, usado para indicar que essa é uma tradução válida.
- *Campo de base.* Equivalente a um endereço de frame de página contendo o endereço físico do primeiro byte do segmento.
- *Bit de acesso.* Semelhante ao bit de referência ou bit de uso em algumas arquiteturas, que é útil para algoritmos de substituição.
- *Campo de atributos.* Especifica as operações válidas e níveis de proteção para as operações que usam esse segmento.

Há também um *campo de limite*, não encontrado nos sistemas paginados, que estabelece o limite superior dos offsets válidos para esse segmento. A Figura B.25 mostra exemplos de descritores de segmento IA-32.

O IA-32 oferece um sistema de paginação opcional, além desse endereçamento segmentado. A parte superior do endereço de 32 bits seleciona o descritor de segmento, e a parte do meio é um índice para a tabela de página selecionada pelo descritor. A seguir, descreveremos o sistema de proteção que não conta com a paginação.

Adicionando compartilhamento e proteção

Para oferecer o compartilhamento protegido, metade do espaço de endereços é compartilhada por todos os processos e metade é exclusiva a cada processo, o que chamamos de

espaço de endereços global e *espaço de endereços local*, respectivamente. Cada metade recebe uma tabela de descritor com o nome apropriado. Um descritor apontando para um segmento compartilhado é colocado na tabela de descritor global, enquanto um descritor para um segmento privado é colocado na tabela de descritor local.

Um programa carrega um registrador de segmento IA-32 com um índice para a tabela *e* um bit dizendo qual tabela ele deseja. A operação é verificada de acordo com os atributos no descritor, o endereço físico é formado pela inclusão do offset no processador à base no descritor, desde que o offset seja menor que o campo de limite. Cada descritor de segmento possui um campo de 2 bits separado para dar o nível de acesso legal desse segmento. Uma violação ocorrerá somente se o programa tentar usar um segmento com um nível de proteção menor no descritor de segmento.

Agora podemos mostrar como chamar o programa de folha de pagamento mencionado anteriormente para atualizar a informação acumulada no ano sem permitir a atualização de salários. O programa poderia receber um descritor para a informação que possui o campo de escrita apagado, significando que pode ler, mas não pode escrever os dados. Então, um programa confiável pode ser fornecido e escrever apenas a informação de valor acumulado no ano. Ele recebe um descritor com o campo de escrita marcado (Fig. B.26). O programa de folha de pagamento chama o código confiável usando um descritor de segmento de código com o campo de conformidade marcado. Essa definição significa que o programa chamado assume o nível de privilégio do código sendo chamado, em vez do nível de privilégio de quem chamou. Logo, o programa de folha de pagamento pode ler os salários e chamar um programa confiável para atualizar os totais acumulados no ano, enquanto o programa de folha de pagamento não pode modificar os salários. Se houver um cavalo de Troia nesse sistema, para ser eficaz ele precisará estar localizado no código confiável, cuja única tarefa é atualizar a informação de valor acumulado no ano. O argumento para esse estilo de proteção é que limitar o escopo da vulnerabilidade melhora a segurança.

Adicionando chamadas seguras do usuário para as portas do SO e nível de proteção para parâmetros

Permitir que o usuário salte para dentro do sistema operacional é um passo corajoso. Como, então, um projetista de hardware pode aumentar as chances de um sistema seguro sem confiar no sistema operacional ou em qualquer outra parte do código? A técnica do IA-32 é restringir onde o usuário pode entrar com um trecho do código, para colocar com segurança os parâmetros na pilha apropriada e certificar-se de que os parâmetros do usuário não recebem o nível de proteção do código que chamou.

Para restringir a entrada no código dos outros, o IA-32 oferece um descritor de segmento especial, ou *porta de chamada*, identificado por um bit no campo de atributos. Diferentemente de outros descritores, as portas de chamada são endereços físicos completos de um objeto na memória; o offset fornecido pelo processador é ignorado. Como já dissemos, sua finalidade é impedir que o usuário salte aleatoriamente para qualquer lugar em um segmento de código protegido ou mais privilegiado. Em nosso exemplo de programação, isso significa que o único lugar onde o programa de folha de pagamento pode invocar o código confiável é no limite apropriado. Essa restrição é necessária para fazer com que os segmentos em conformidade funcionem como desejado.

O que acontece se quem chama e quem é chamado forem "mutuamente desconfiados", de modo que um não confia no outro? A solução é encontrada no campo de contador de palavras no descritor de baixo, na Figura B.26. Quando uma instrução de chamada invoca um descritor de porta de chamada, esse descritor copia o número de palavras especificado no descritor da pilha local para a pilha correspondente ao nível desse segmento.

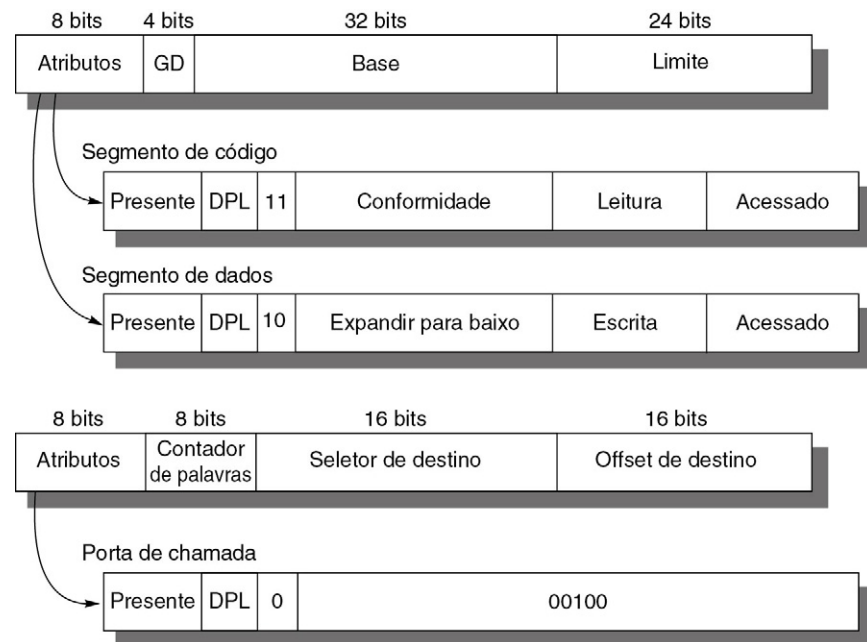



FIGURA B.26 **Os descritores de segurança do IA-32 são distinguidos por bits no campo de atributos.** *Base*, *limite*, *presente*, *leitura* e *escrita* são autoexplicativos. D indica o tamanho de endereçamento default das instruções: 16 bits ou 32 bits. G indica a granularidade do limite de segmento: 0 significa em bytes e 1 significa em páginas de 4 KB. G é definido como 1 quando a paginação é ativada para definir o tamanho das tabelas de página. DPL significa *Descriptor Privilege Level* (nível de privilégio do descritor) — este é verificado contra o nível de privilégio de código para ver se o acesso será permitido. *Conformidade* diz que o código assume o nível de privilégio do código sendo chamado, em vez do nível de privilégio de quem chamou, usado para rotinas de biblioteca. O *campo expandido para baixo* inverte a verificação para permitir que o campo de base seja a marca de limite superior e o campo de limite seja a marca de limite inferior. Como é de se esperar, isso é usado para segmentos de pilha que crescem para baixo. O *contador de palavras* controla o número de palavras copiadas da pilha atual para a nova pilha em uma porta de chamada. Os dois outros campos do descritor de porta de chamada, *seletor de destino* e *offset de destino*, selecionam o descritor do destino da chamada e seu offset, respectivamente. Há muito mais do que esses três descritores de segmento no modelo de proteção IA-32.

A cópia permite que o usuário passe parâmetros colocando-os primeiro na pilha local. O hardware, então, os transfere com segurança para a pilha correta. Um retorno de uma porta de chamada removerá os parâmetros das pilhas e copiará quaisquer valores de retorno para a pilha apropriada. Observe que esse modelo é incompatível com a atual prática de passar parâmetros nos registradores.

Esse esquema ainda deixa aberta a possível brecha de ter o sistema operacional usando o endereço do usuário, passado como parâmetros, com o nível de segurança do sistema operacional em vez do nível de segurança do usuário. O IA-32 resolve esse problema dedicando 2 bits a cada registrador de segmento do processador ao *nível de proteção solicitado*. Quando uma rotina do sistema operacional é invocada, ela pode executar uma instrução que define esse campo de 2 bits em todos os parâmetros de endereço com o nível de proteção do usuário que chamou a rotina. Assim, quando esses parâmetros de endereço são carregados nos registradores de segmento, eles definem o nível de proteção solicitado com o valor apropriado. O hardware do IA-32, então, utiliza o nível de proteção solicitado para impedir qualquer tolice: nenhum segmento poderá ser acessado a partir da rotina do sistema usando esses parâmetros se tiver um nível de proteção mais privilegiado do que o solicitado.

Um exemplo de memória virtual paginada: o gerenciamento de memória de 64 bits do Opteron

Os engenheiros da AMD encontraram poucos usos para o modelo de proteção elaborado que acabamos de descrever. O modelo popular é um espaço de endereços plano, de 32 bits, introduzido pelo 80386, que define todos os valores de base dos registradores de segmento como zero. Daí a AMD ter dispensado os segmentos múltiplos no modo de 64 bits. Ela assume que a base do segmento é zero e ignora o campo de limite. Os tamanhos de página são de 4 KB, 2 MB e 4 MB.

O endereço virtual de 64 bits da arquitetura do AMD64 é mapeada nos endereços físicos de 52 bits, embora as implementações possam implementar menos bits para simplificar o hardware. O Opteron, por exemplo, utiliza endereços virtuais de 48 bits e endereços físicos de 40 bits. O AMD64 exige que os 16 bits mais significativos do endereço virtual sejam apenas a extensão de sinal dos 48 bits inferiores, o que é chamado *forma canônica*.

O tamanho das tabelas de página para o espaço de endereços de 64 bits é alarmante. Logo, o AMD64 utiliza uma tabela de página hierárquica multinível para mapear o espaço de endereços, a fim de manter um tamanho razoável. O número de níveis depende do tamanho do espaço de endereços virtuais. A Figura B.27 mostra a tradução de quatro níveis dos endereços virtuais de 48 bits do Opteron.

Os offsets para cada uma dessas tabelas de página vêm dos quatro campos de 9 bits. A tradução de endereço começa com a adição do primeiro offset ao registro de base 4 em nível de mapa de página e depois lendo a memória desse local para obter a base da tabela de página do nível seguinte. O próximo offset de endereço é, por sua vez, somado a esse endereço recém-apanhado, e a memória é acessada novamente para determinar a base da terceira tabela de

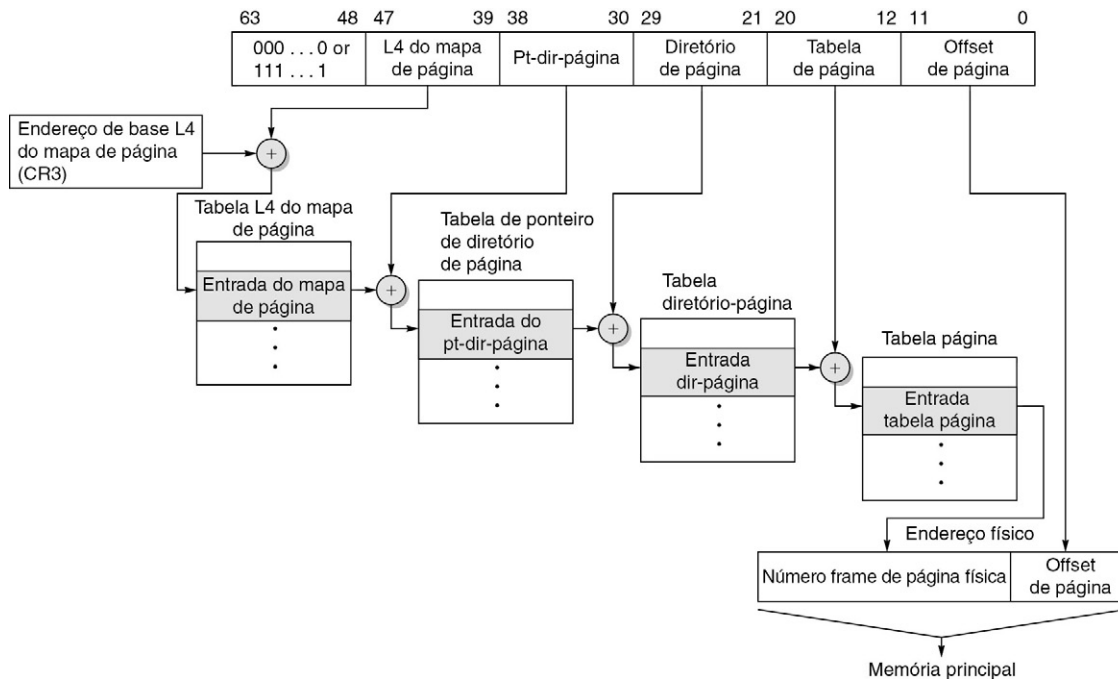



FIGURA B.27 **Mapeamento de um endereço virtual do Opteron.**
A implementação de memória virtual do Opteron com quatro níveis de tabela de página admite um tamanho de endereço físico efetivo de 40 bits. Cada tabela de página possui 512 entradas, de modo que cada campo de nível tem 9 bits de largura. A arquitetura AMD64 permite que o tamanho do endereço cresça dos atuais 48 bits para 64 bits, e o tamanho do endereço físico cresça dos atuais 40 bits para 52 bits.

página. Isso acontece novamente da mesma maneira. O último campo de endereço é somado a esse endereço de base final, e a memória é lida usando essa soma para (finalmente) apanhar o endereço físico da página sendo referenciada. Esse endereço é concatenado ao offset de página de 12 bits para obter o endereço físico completo. Observe que a tabela de página na arquitetura do Opteron se ajusta dentro de uma única página de 4 KB.

O Opteron usa uma entrada de 64 bits em cada uma dessas tabelas de página. Os 12 primeiros bits são reservados para uso futuro, os próximos 52 bits contêm o número do frame da página física e os últimos 12 bits mostram a informação de proteção e uso. Embora os campos variem um pouco entre os níveis da tabela de página, aqui estão os básicos:

- *Presença.* Diz que a página está presente na memória.
- *Leitura/escrita.* Diz se a página é apenas de leitura ou de leitura/escrita.
- *Usuário/supervisor.* Diz se um usuário pode acessar a página ou se ela é limitada aos três níveis de privilégio superiores.
- *Modificação.* Diz se a página foi modificada.
- *Acessado.* Diz se a página foi lida ou escrita desde que o bit foi apagado pela última vez.
- *Tamanho de página.* Diz se o último nível é para páginas de 4 KB ou páginas de 4 MB; se for o segundo, então o Opteron só usa três em vez de quatro níveis de páginas.
- *Não executar.* Não encontrado no esquema de proteção do 80386, esse bit foi acrescentado para impedir que o código seja executado em algumas páginas.
- *Desativação da cache em nível de página.* Diz se a página pode ser colocada em cache ou não.
- *Write-through em nível de página.* Diz se a página permite write-back ou write-through para as caches de dados.

Como o Opteron normalmente passa por quatro níveis de tabelas em uma falta no TLB, existem três lugares em potencial para verificar as restrições de proteção. O Opteron obedece apenas à entrada da tabela de página (Page Table Entry — PTE) de nível inferior, verificando as outras somente para ter certeza de que o bit de validade está marcado.

Como a entrada tem 8 bytes de extensão, cada tabela de página possui 512 entradas, e o Opteron possui 4 KB de páginas; as tabelas de página possuem exatamente uma página de extensão. Cada um dos quatro campos de nível possui 9 bits de extensão e o offset de página possui 12 bits. Essa derivação deixa $64 - (4 \times 9 + 12)$ ou 16 bits para serem estendidos por sinal, a fim de garantir endereços canônicos.

Embora tenhamos explicado a tradução válida dos endereços, o que impede que o usuário crie traduções de endereço ilegais e faça uma bagunça? As próprias tabelas de página são protegidas contra escrita pelos programas do usuário. Assim, o usuário pode experimentar qualquer endereço virtual, mas, controlando as entradas da tabela de página, o sistema operacional controla qual memória física é acessada. O compartilhamento da memória entre os processos é obtido fazendo-se com que uma entrada de tabela de página em cada espaço de endereço aponte para a mesma página de memória física.

O Opteron emprega quatro TLBs para reduzir o tempo de tradução de endereço: dois para acessos de instrução e dois para acessos de dados. Assim como as caches multiníveis, o Opteron reduz as faltas de TLB tendo dois TLBs L2 maiores: um para instruções e um para dados. A Figura B.28 descreve o TLB de dados.

Parâmetro	Descrição
Tamanho de bloco	1 PTE (8 bytes)
Tempo de acerto L1	1 ciclo de clock
Tempo de acerto L2	7 ciclos de clock
Tamanho do TLB L1	O mesmo para TLBs de instrução e dados: 40 PTEs por TLB, com 32 páginas de 4 KB e 8 para páginas de 2M ou 4M
Tamanho do TLB L2	O mesmo para TLBs de instrução e dados: 512 PTEs de páginas de 4 KB
Seleção de bloco	LRU
Estratégia de escrita	(Não se aplica)
Alocação do bloco L1	Totalmente associativo
Alocação do bloco L2	Associativo por conjunto com 4 vias

FIGURA B.28 Parâmetros de hierarquia da memória dos TLBs de instrução e dados L1 e L2 do Opteron.

Resumo: proteção no Intel Pentium de 32 bits contra AMD Opteron de 64 bits

O gerenciamento de memória no Opteron é típico da maioria dos computadores desktops e servidores de hoje, contando com a tradução de endereço em nível de página e a operação correta do sistema operacional para fornecer segurança a múltiplos processos que compartilham o computador. Embora apresentadas como alternativas, a Intel seguiu o caminho da AMD e abraçou a arquitetura AMD64. Logo, tanto AMD quanto Intel admitem uma extensão de 64 bits do 80x86, embora, por motivos de compatibilidade, ambos admitam o esquema de proteção segmentado, mais elaborado.

Se o modelo de proteção segmentado parece mais difícil de ser montado do que o modelo AMD64, isso é porque realmente ele é. Esse esforço deverá ser especialmente frustrante para os engenheiros, pois poucos clientes utilizam o mecanismo de proteção elaborado. Além disso, o fato de o modelo de proteção ser divergente da proteção de página simples dos sistemas tipo UNIX, significa que ele será usado apenas por alguém que esteja escrevendo um sistema operacional especialmente para esse computador, o que ainda não aconteceu.

B.6 FALÁCIAS E ARMADILHAS

Até mesmo uma revisão sobre hierarquia de memória possui falácias e armadilhas!

Armadilha. *Um espaço de endereços muito pequeno.*

Apenas cinco anos depois que a DEC e a Carnegie Mellon University colaboraram para projetar uma nova família de computadores PDP-11, ficou nítido que sua criação tinha um erro fatal. Uma arquitetura anunciada pela IBM seis anos *antes* do PDP-11 ainda estava florescendo, com pequenas modificações, 25 anos depois. E o VAX da DEC, criticado por incluir funções desnecessárias, vendeu milhões de unidades depois que o PDP-11 saiu de produção. Por quê?

O erro fatal do PDP-11 foi o tamanho de seus endereços (16 bits) em comparação com os tamanhos de endereço do IBM 360 (24 a 31 bits) e do VAX (32 bits). O tamanho do endereço limita a extensão do programa, pois o tamanho de um programa e a quantidade de dados necessários pelo programa precisam ser menores que $2^{\text{Tamanho de endereço}}$. O motivo

do tamanho de endereço ser tão difícil de mudar é que ele determina a largura mínima de qualquer coisa que possa conter um endereço: PC, registrador, palavra da memória e aritmética do endereço efetivo. Se não houver um plano para expandir o endereço do início, então as chances de alterar o tamanho do endereço com sucesso são tão pequenas que isso normalmente significa o final dessa família de computadores. Bell e Strecker (1976) colocam isso desta forma:

> Há somente um erro que pode ser cometido no projeto de computador que é difícil de se recuperar — não tem bits de endereço suficientes para o endereçamento e gerenciamento de memória. O PDP-11 seguiu a tradição imutável de quase todo computador conhecido. [p. 2]

A lista parcial de computadores bem-sucedidos que lutaram até o fim pela falta de bits de endereço inclui PDP-8, PDP-10, PDP-11, Intel 8080, Intel 8086, Intel 80186, Intel 80286, Motorola 6800, AMI 6502, Zilog Z80, CRAY-1 e CRAY X-MP.

A venerável linha 80x86 leva a distinção de ter sido estendida duas vezes, primeiro para 32 bits com o Intel 80386, em 1985, e recentemente para 64 bits, com o AMD Opteron.

Armadilha. *Ignorar o impacto do sistema operacional sobre o desempenho da hierarquia de memória.*

A Figura B.29 mostra o tempo de stall da memória devido ao sistema operacional gasto em três cargas de trabalho grandes. Cerca de 25% do tempo de stall é gasto em faltas no sistema operacional ou resulta de faltas nos programas de aplicação, devido a interferências no sistema operacional.

Armadilha. *Contar com os sistemas operacionais para alterar o tamanho da página com o tempo.*

Os arquitetos do Alpha tinham um plano elaborado para fazer crescer a arquitetura com o tempo, aumentando seu tamanho de página, até mesmo com base no tamanho de seu endereço virtual. Quando chegou o momento de aumentar os tamanhos de página com os últimos Alphas, os projetistas de sistema operacional se frustraram e o sistema de

			Tempo						
Faltas			% Tempo devido a faltas da aplicação		% Tempo devido diretamente a faltas do SO				
Carga de trabalho	% nas aplicações	% no SO	Faltas inerentes à aplicação	Conflitos do SO com as aplicações	Faltas de instrução do SO	Faltas de dados para migração	Faltas de dados nas operações em bloco	Restante das faltas do SO	% Tempo de faltas do SO e conflitos da aplicação
Pmake	47%	53%	14,1%	4,8%	10,9%	1,0%	6,2%	2,9%	25,8%
Multipgm	53%	47%	21,6%	3,4%	9,2%	4,2%	4,7%	3,4%	24,9%
Oracle	73%	27%	25,7%	10,2%	10,6%	2,6%	0,6%	2,8%	26,8%

FIGURA B.29 **Faltas e tempo gasto nas faltas para aplicações e o sistema operacional.**
O sistema operacional aumenta cerca de 25% o tempo de execução da aplicação. Cada processador possui uma cache de instruções de 64 KB e uma cache de dados de dois níveis com 64 KB no primeiro nível e 256 KB no segundo nível; todas as caches são mapeadas diretamente com blocos de 16 bytes. Coletados na estação Silicon Graphics POWER 4D/340, um multiprocessador com quatro processadores R3000 de 33 MHz rodando três cargas de trabalho de aplicação sob um UNIX System V-Pmake: uma compilação paralela de 56 arquivos; Multipgm: o programa numérico paralelo MP3D rodando simultaneamente com Pmake e uma sessão de edição de cinco telas; e Oracle: rodando uma versão restrita do benchmark TP-1 usando o banco de dados Oracle. *(Dados de Torrellas, Gupta e Hennessy, 1992.)*

memória virtual foi revisado para aumentar o espaço de endereços enquanto mantinha a página de 8 KB.

Os arquitetos de outros computadores observaram taxas de falta de TLB muito altas, e por isso acrescentaram múltiplos tamanhos de página ao TLB. A esperança foi de que os programadores de sistema operacional alocassem na maior página um objeto que fizesse sentido, preservando assim as entradas de TLB. Após uma década de tentativa, a maioria dos sistemas operacionais utiliza essas "superpáginas" somente para funções escolhidas a dedo: mapear a memória de vídeo ou outros dispositivos de E/S ou usar páginas muito grandes para o código de banco de dados.

B.7 COMENTÁRIOS FINAIS

A dificuldade de se montar um sistema de memória para acompanhar o ritmo dos processadores mais rápidos é enfatizada pelo fato de que o material bruto para a memória principal é o mesmo encontrado no computador mais barato. É o princípio da localidade que nos ajuda aqui — sua firmeza é demonstrada em todos os níveis da hierarquia da memória nos computadores atuais, de discos a TLBs.

Porém, a latência relativa crescente para a memória, levando centenas de ciclos de clock em 2011, significa que os programadores e escritores de compilador precisam estar cientes dos parâmetros das caches e TLBs, se quiserem que seus programas funcionem bem.

B.8 PERSPECTIVAS HISTÓRICAS E REFERÊNCIAS

Na Seção L.3 (disponível on-line), examinamos a história das caches, memória virtual e máquinas virtuais (a seção histórica cobre este apêndice e o Capítulo 3). A IBM desempenha um papel de destaque nessa história. Também estão incluídas referências para leitura adicional.

EXERCÍCIOS POR AMR ZAKY

B.1 [10/10/10/15] <B.1> Você está tentando apreciar a importância do princípio da localidade para justificar o uso de uma memória de cache, e experimenta isso usando um computador com uma cache de dados L1 e uma memória principal (você se concentra exclusivamente nos acessos de dados). As latências (em ciclos de CPU) dos diferentes tipos de acessos são as seguintes: acerto na cache, 1 ciclo; falta na cache, 105 ciclos; acesso à memória principal com a cache desabilitada, 100 ciclos.

- **a.** [10] <B.1> Quando você executa um programa com taxa de falta geral de 5%, qual será o tempo de acesso à memória geral (em ciclos de CPU)?
- **b.** [10] <B.1> A seguir, você executa um programa projetado especificamente para produzir dados de endereço completamente aleatórios sem localidade. Para isso, você usa um array de tamanho 256 MB (ele cabe inteiro na memória principal). Acessos a elementos aleatórios desse array são feitos continuamente (usando um gerador uniforme de números aleatórios para gerar os índices de elementos). Se o tamanho do sua cache de dados for de 64 KB, qual será o tempo médio de acesso à memória?
- **c.** [10] <B.1> Se você comparar o resultado obtido no item *b* com o tempo de acesso à memória principal quando a cache estiver desabilitada, o que pode concluir sobre o papel do princípio da localidade para justificar o uso de memória de cache?

d. [15] <B.1> Você observou que um acerto na cache produz um ganho de 99 ciclos (1 ciclo *versus* 100), mas produz uma perda de 15 ciclos no caso de uma falta (105 ciclos *versus* 100). No caso geral, podemos expressar essas duas quantidades como G (ganho) e L (perda — *loss*). Usando essas duas quantidades (G e L), identifique a taxa de falta mais alta depois da qual o uso da cache não seria vantajoso.

B.2 [15/15] <B.1> Para fins deste exercício, consideramos que temos uma cache de 512 bytes com blocos de 64 bytes. Consideramos também que a memória principal tem 2 KB. Podemos considerar a memória como um array de blocos de 64 bytes: M0, M1, …, M31. A Figura B.30 delineia os blocos de memória que podem residir em diferentes blocos de cache se a cache fosse totalmente associativa.

a. [15] <B.1> Mostre o conteúdo da tabela se a cache for organizada como uma cache mapeada diretamente.

b. [15] <B.1> Repita o item *a* com a cache organizada coma cache associativa por conjunto com quatro vias.

B.3 [10/10/10/10/15/10/15/20] <B.1> Muitas vezes, a organização da cache é influenciada pelo desejo de reduzir o consumo de energia da cache. Para isso, supomos que a cache é distribuída fisicamente em um array de dados (contendo os dados), array de tags (contendo as tags) e array de substituição (contendo informações necessárias para a política de substituição). Além do mais, cada um desses arrays é distribuído fisicamente em múltiplos subarrays (um por via), que podem ser acessados individualmente. Por exemplo, uma cache associativa por conjunto com quatro vias usada menos recentemente (Least Recently Used — LRU) teria quatro subarrays de dados, quatro subarrays de tags e quatro subarrays de substituição. Nós supomos que os subarrays de substituição são alcançados uma vez por acesso quando a política de substituição LRU for usada, e uma vez por falta se a política de substituição primeiro a entrar, primeiro a sair (FIFO) for usada. Isso não é necessário quando se utiliza uma política de substituição aleatória. Para uma cache específica, foi determinado que os acessos aos diferentes arrays têm os seguintes pesos de consumo energético:

Array	Peso do consumo de energia (por via acessada)
Array de dados	20 unidades
Array de tags	5 unidades
Array de miscelânea	1 unidade

Bloco de cache	Conjunto	Via	Blocos de memória que podem residir dentro do bloco de cache
0	0	0	M0, M1, M2, …, M31
1	0	1	M0, M1, M2, …, M31
2	0	2	M0, M1, M2, …, M31
3	0	3	M0, M1, M2, …, M31
4	0	4	M0, M1, M2, …, M31
5	0	5	M0, M1, M2, …, M31
6	0	6	M0, M1, M2, …, M31
7	0	7	M0, M1, M2, …, M31

FIGURA B.30 Blocos de memória que podem residir dentro do bloco de cache.

Estime o uso de energia da cache (em unidades de potência) para as seguintes configurações. Nós supomos que a cache é associativa por conjunto com quatro vias. O consumo de energia de acesso à memória principal — embora importante — não é considerado aqui. Dê resposta para as políticas de substituição LRU, FIFO e aleatório.

a. [10] <B.1> Um acerto na leitura de cache. Todos os arrays são lidos simultaneamente.

b. [10] <B.1> Repita o item *a* para uma falta na leitura de cache.

c. [10] <B.1> Repita o item *a* supondo que o acesso à cache seja dividido entre dois ciclos. No primeiro ciclo, todos os subarrays de tag são acessados. No segundo ciclo, somente o subarray correspondente será acessado.

d. [10] <B.1> Repita o item *c* para uma falta na leitura de cache (não há acessos ao array de dados no segundo ciclo).

e. [15] <B.1> Repita o item *c* supondo que seja adicionada uma lógica para prever a via da cache a ser acessada. Somente o subarray de tags da via prevista é acessada no ciclo um. Um acerto na via (endereços correspondendo na via prevista) implica um acerto na cache. Uma falta na via dita o exame de todos os subarrays de tag no segundo ciclo. No caso de um acerto na via, somente um subarray de dados (aquele em que houve correspondência de tag) é acessado no ciclo dois. Suponha que há um acerto na via.

f. [10] <B.1> Repita o item *e* supondo que o previsor de via errou (a via que ele escolheu está errada). Quando ele falha, o previsor de via adiciona um ciclo extra no qual ele acessa todos os subarrays de tag. Suponha um acerto na leitura de cache.

g. [15] <B.1> Repita o item *f* supondo uma falta na leitura de cache.

h. [20] <B.1> Use os itens *e*, *f* e *g* para o caso geral em que a carga de trabalho tenha as seguintes estatísticas: taxa de falta do previsor de via = 5% e taxa de falta de cache = 3% (considere diferentes políticas de substituição).

B.4[10/10/15/15/15/20] <B.1> Nós comparamos os requisitos de largura de banda de escrita do write-through em comparação com o write-back usando um exemplo concreto. Vamos supor que temos uma cache de 64 KB com tamanho de linha de 32 bytes. A cache vai alocar uma linha ao ocorrer uma falta na escrita. Se configurada como cache write-back, ela vai realizar o write-back de toda a linha modificada se ela precisar ser substituída. Vamos também supor que a cache seja conectada ao nível inferior na hierarquia através de um barramento com 64 bits (8 bytes) de largura. O número de ciclos de CPU para um acesso de escrita de B bytes neste barramento é

$$10+5\left(\left\lceil \frac{B}{8} \right\rceil - 1\right)$$

Por exemplo, uma escrita de 8 bytes levaria 10 + 5($\lceil 8/8 \rceil$ − 1) ciclos, enquanto usar a mesma fórmula em uma escrita de 12 bytes levaria 15 ciclos. Responda às seguintes perguntas consultando o trecho de código C a seguir:

…

```
#define PORTION 1 … Base = 8*i; for (unsigned int j=base;
j < base+PORTION; j++) //Suponha que j seja armazenado em um registrador
data[j] = j;
```

a. [10] <B.1> Para uma cache write-through, quantos ciclos de CPU são gastos em transferências de escrita para a memória para todas as iterações combinadas do loop `j`?

b. [10] <B.1> Se a cache for configurada como uma cache write-back, quantos ciclos de CPU são gastos no write-back de uma linha de cache?

c. [15] <B.1> Mude `PORTION` para 8 e repita o item *a*.

d. [15] <B.1> Qual é o número mínimo de atualizações de array para a mesma linha de cache (antes de substituí-la) que tornaria a cache write-back superior?

e. [15] <B.1> Pense em um cenário em que todas as palavras da linha de cache serão escritas (não necessariamente usando o código anterior), e uma cache write-through vai precisar de menos ciclos totais de CPU do que a cache write-back.

B.5 [10/10/10/10] <B.2> Você está construindo um sistema ao redor de um processador com execução em ordem que roda a 1,1 GHz e tem CPI de 0,7, excluindo os acessos à memória. As únicas instruções que leem ou escrevem dados na memória são loads (20% de todas as instruções) e stores (5% de todas as instruções). O sistema de memória para esse computador é composto de uma cache L1 dividida, que não impõe penalidade para os acertos. A cache I e a cache D são mapeadas diretamente e contêm 32 KB cada uma. A cache I tem taxa de falta de 2% e blocos de 32 bytes, e a cache D é write-through, com taxa de falta de 5% e blocos de 16 bytes. Existe um buffer de escrita na cache D que elimina stalls para 95% de todas as escritas. A cache L2 write-back e unificada de 512 KB tem blocos de 64 bytes e tempo de acesso de 15 ns. Ela é conectada à cache L1 através de um barramento de dados de 128 bits que roda a 266 MHz e pode transferir uma palavra de 128 bits por ciclo de barramento. De todas as referências de memória enviadas para a cache L2 nesse sistema, 80% são satisfeitas sem ir para a memória principal. Além disso, 50% de todos os blocos substituídos são modificados. A memória principal, com 128 bits de largura, tem latência de acesso de 60 ns, depois do que o número de palavras no barramento pode ser transferida à taxa de uma por ciclo no barramento de memória principal de 133 MHz, com 128 bits de largura.

a. [10] <B.2> Qual é o tempo médio de acesso à memória por acesso de instrução?

b. [10] <B.2> Qual é o tempo médio de acesso à memória por leitura de dado?

c. [10] <B.2> Qual é o tempo médio de acesso à memória por escrita de dado?

d. [10] <B.2> Qual é o CPI geral, incluindo os acessos de memória?

B.6 [10/15/15] <B.2> Converter a taxa de falta (faltas por referência) em faltas por instrução depende de dois fatores: referências por instrução buscada e a fração de instruções buscadas que realmente são confirmadas.

a. [10] <B.2> A fórmula para faltas por instrução na página 488 é escrita primeiro em termos de três fatores: taxa de falta, acessos à memória e número de instruções. Cada um desses fatores representa eventos reais. O que muda nas faltas de escrita por instrução como tempos de taxa de falta vezes o fator *acessos à memória por instrução*?

b. [15] <B.2> Processadores especulativos vão buscar instruções que não são confirmadas. A fórmula para faltas por instrução na página 488 refere-se a faltas por instrução no caminho de execução, ou seja, somente as instruções que devem realmente ser executadas para realizar o programa. Converta a fórmula para faltas por instrução, na página 488, naquela que usa somente a taxa de faltas, referências por instrução buscada e fração das instruções buscadas que são confirmadas. Por que depender desses fatores em vez daqueles na fórmula na página 488?

c. [15] <B.2> A conversão no item *b* poderia gerar um valor incorreto no sentido de que o valor do fator referências por instrução buscada não é igual ao número de referências para qualquer instrução particular. Reescreva a fórmula do item *b* para corrigir essa deficiência.

B.7 [20] <B.1, B.3> Em sistemas com cache L1 write-through suportada por uma cache L2 write-back em vez da memória principal, um buffer merging de escrita pode ser simplificado. Explique como isso pode ser feito. Nessas situações, ter um buffer de escrita total (em vez da versão simples que você acabou de propor) poderia ser útil?

B.8 [20/20/15/25] <B.3> A política LRU de substituição baseia-se na suposição de que só o endereço A1 foi acessado menos recentemente do que o endereço A2, então, no futuro, A2 será acessado novamente antes de A1. Portanto, A2 recebe prioridade sobre A1. Discuta como essa suposição não se mantém quando um loop maior do que a cache de instrução está sendo executado continuamente. Considere, por exemplo, uma cache de instruções totalmente associativa de 128 bytes com um bloco de 4 bytes (cada bloco pode conter exatamente uma instrução). A cache usa uma política LRU de substituição.

a. [20] <B.3> Qual é a taxa de falta assintótica para um loop de 64 bytes com grande número de iterações?

b. [20] <B.3> Repita o item *a* para tamanhos de loop de 192 bytes e 320 bytes.

c. [15] <B.3> Se a política de substituição de cache for mudada para mais recentemente usada (MRU) (substitui a linha de cache acessada mais recentemente), qual dos três casos anteriores (loops de 64, 192 ou 320 bytes) se beneficiaria dela?

d. [25] <B.3> Sugira políticas de substituição adicionais que podem ter desempenho melhor do que a LRU.

B.9 [20] <B.3> Aumentar a associatividade de uma cache (com os demais parâmetros mantidos constantes), reduz estatisticamente a taxa de falta. Entretanto, pode haver casos patológicos em que aumentar a associatividade de uma cache vai aumentar a taxa de falta para uma carga de trabalho particular. Considere o caso do mapeamento direto em comparação com uma cache associativa por conjunto com duas vias de tamanho igual. Considere que a cache associativa por conjunto usa a política de substituição LRU. Para simplificar, suponha que o tamanho de bloco seja uma palavra. Agora mostre a sequência de acessos de palavra que vai produzir mais faltas na cache associativa com duas vias. (*Dica*: Concentre-se em construir uma sequência de acessos que sejam direcionados exclusivamente a um único conjunto da cache associativa por conjunto de duas vias, de tal modo que a mesma sequência acesse exclusivamente dois blocos na cache mapeada diretamente.)

B.10 [10/10/15] <B.3> Considere uma hierarquia de memória de dois níveis composta de caches de dados L1 e L2. Suponha que as duas caches usem a política write-back em um acerto na escrita e que as duas tenham o mesmo tamanho de bloco. Liste as ações realizadas em resposta aos seguintes eventos:

a. [10] <B.3> Uma falta na cache L1 quando as caches são organizadas em uma hierarquia inclusiva.

b. [10] <B.3> Uma falta na cache L1 quando as caches são organizadas em uma hierarquia exclusiva.

c. [15] <B.3> Nos itens *a* e *b*, considere a possibilidade de que a linha retirada pode ser não modificada ou modificada.

B.11 [15/20] <B.2, B.3> Impedir algumas instruções de entrar na cache pode reduzir as faltas por conflito.

a. [15] <B.3> Esboce uma hierarquia de programa em que é melhor impedir que partes do programa entrem na cache de instruções. (*Dica*: Considere um programa com blocos de código que são colocados em ninhos de loop mais profundos do que outros blocos.)

b. [20] <B.2, B.3> Sugira técnicas de software ou hardware para implantar a exclusão de certos blocos da cache de instruções.

B.12 [15] <B.4> Um programa está sendo executado em um computador com um translation lookaside buffer (TLB) totalmente associativo (micro) de quatro entradas.

VP#	PP#	Entrada válida
5	30	1
7	1	0
10	10	1
15	25	1

O que mostramos a seguir é a sequência de números de página virtual acessados por um programa. Para cada acesso, indique se ele produz um acerto/falta no TLB e, se ele acessar a tabela da página, se produz um acerto ou falta na página. Faça um X na coluna da tabela de página se ela não for acessada.

Índice de página virtual	N.° da página física	Presente
0	3	S
1	7	N
2	6	N
3	5	S
4	14	S
5	30	S
6	26	S
7	11	S
8	13	N
9	18	N
10	10	S
11	56	S
12	110	S
13	33	S
14	12	N
15	25	S

Página virtual acessada	TLB (acerto ou falta)	Tabela de página (acerto ou falta)
1		
5		
9		
14		
10		
6		
15		
12		
7		
2		

B.13 [15/15/15/15] <B.4> Alguns sistemas de memória lidam com faltas TLB no software (como exceção), enquanto outros usam hardware para faltas de TLB.

a. [15] <B.4> Quais são os trade-offs entre esses dois métodos de lidar com as faltas TLB?

b. [15] <B.4> O tratamento das faltas TLB no software será sempre mais lento do que tratar as faltas TLB por hardware? Explique.

c. [15] <B.4> Existem estruturas de tabela de página que seriam difíceis de tratar no hardware, mas que são possíveis no software? Existem estruturas que seriam difíceis de o software tratar, porém fáceis para o hardware?

d. [15] <B.4> Por que as taxas de falta TLB para programas de ponto flutuante são geralmente mais altas do que aquelas para programas de inteiros?

B.14 [25/25/25/25/20] <B.4> Qual deve ser o tamanho de um TLB? As faltas de TLB geralmente são muito rápidas (menos de 10 instruções mais o custo de uma exceção), então pode não valer a pena ter um grande TLB apenas para reduzir um pouco a taxa de faltas TLB. Usando o simulador SimpleScalar (www.cs.wisc.edu/~mscalar/simplescalar.html) e um ou mais benchmarks SPEC95, calcule a taxa de falta TLB e o overhead TLB (em porcentagem de tempo gasto lidando com faltas TLB) para as seguintes configurações de TLB. Suponha que cada falta na TLB exija 20 instruções.

a. [25] <B.4> 128 entradas, associativa por conjunto com duas vias, páginas de 4 KB a 64 KB (progredindo em potências de 2).

b. [25] <B.4> 256 entradas, associativa por conjunto com duas vias, páginas de 4 KB a 64 KB (progredindo em potências de 2).

c. [25] <B.4> 512 entradas, associativa por conjunto com duas vias, páginas de 4 KB a 64 KB (progredindo em potências de 2).

d. [25] <B.4> 1.024 entradas, associativa por conjunto com duas vias, páginas de 4 KB a 64 KB (progredindo em potências de 2).

e. [20] <B.4> Qual seria o efeito de um ambiente multitarefa sobre a taxa de falta de TLB e o overhead? Como a frequência de troca de contexto afetaria o overhead?

B.15 [15/20/20] <B.5> É possível proporcionar uma proteção mais flexível que a da arquitetura Intel Pentium usando um esquema de proteção similar ao usado na arquitetura Precision da Hewlett-Packard (HP/PA). Em um esquema assim, cada entrada da tabela de página contém uma "ID de proteção" (chave) juntamente com os direitos de acesso para a página. Em cada referência, a CPU compara a ID de proteção na entrada da tabela de página com aquelas armazenadas em cada um dos quatro registradores de ID de proteção (o acesso a esses registradores requer que a CPU esteja em modo de supervisor). Se não houver correspondência para a ID de proteção na entrada da tabela de página ou se o acesso não for um acesso permitido (escrever em uma página somente para leitura, por exemplo), será gerada uma exceção.

a. [15] <B.5> Como um processo poderia ter mais de quatro IDs de proteção válidos a quaisquer dados em determinado momento? Em outras palavras, suponha que um processo desejasse ter 10 IDs de proteção simultaneamente. Proponha um mecanismo pelo qual isso pudesse ser feito (talvez com a ajuda do software).

b. [20] <B.5> Explique como esse modelo poderia ser usado para facilitar a construção de sistemas operacionais a partir de trechos de código relativamente pequenos que não possam se sobrescrever um ao outro (microkernels). Que vantagens tal sistema operacional poderia ter sobre um sistema operacional monolítico no qual qualquer código no SO pode escrever em qualquer local da memória?

c. [20] <B.5> Uma simples mudança de projeto para esse sistema permitiria duas IDs de proteção para cada entrada de tabela de página, uma para acessos de leitura e outra para acessos de escrita ou execução (o campo não será usado, se nem o bit de escrita nem o bit execução estiverem setados). Que vantagens poderia haver em ter diferentes IDs de proteção para os recursos de leitura e escrita? (*Dica*: Isso poderia tornar mais fácil compartilhar dados e código entre processadores?)

Pipelining: conceitos básicos e intermediários

Esse é bem um problema dos três cachimbos.[1]

Sir Arthur Conan Doyle
As aventuras de Sherlock Holmes

C.1 INTRODUÇÃO

Muitos leitores deste apêndice já devem ter visto os fundamentos do pipelining em outro lugar (como em nosso texto mais básico, *Organização e projeto de computador*), em outro curso. Como o Capítulo 3 trabalha bastante com esse material, antes de prosseguir os leitores deverão garantir que estejam familiarizados com os conceitos discutidos aqui. Ao ler o Capítulo 2, você poderá achar útil voltar a este material para fazer uma rápida revisão.

Iniciamos o apêndice com os fundamentos do pipelining, incluindo a discussão sobre implicações do datapath, uma introdução sobre hazards e o exame do desempenho dos pipelines. Esta seção descreve o pipeline RISC básico em cinco estágios, que é a base para o restante do apêndice. A Seção C.2 descreve a questão dos hazards, por que eles causam problemas de desempenho e como podem ser tratados. A Seção C.3 discute como o pipeline simples em cinco estágios é realmente implementado, enfocando o controle e o modo como os hazards são tratados.

A Seção C.4 analisa a interação entre o pipelining e diversos aspectos do projeto do conjunto de instruções, incluindo uma discussão dos tópicos importantes das exceções e sua interação com o pipelining. Os leitores não familiarizados com os conceitos das interrupções precisas e imprecisas e a retomada após as exceções acharão esse material útil, pois são a chave para entender as técnicas mais avançadas do Capítulo 3.

A Seção C.5 mostra como o pipeline de cinco estágios pode ser estendido para lidar com instruções de ponto flutuante, que têm maior duração. A Seção C.6 reúne esses conceitos

[1]**Nota da Tradução:** *Three-pipe problem*: trocadilho em inglês com a palavra *pipe*, também usada em *pipeline*.

em um estudo de caso de um processador com pipeline de maior profundidade, o MIPS R4000/4400, que inclui o pipeline de inteiros com oito estágios e o pipeline de ponto flutuante.

A Seção C.7 introduz o conceito de escalonamento dinâmico e o uso de scoreboards para implementar o escalonamento dinâmico. Ele é apresentado como uma questão cruzada, pois pode ser usado para servir de introdução aos conceitos centrais do Capítulo 3, que enfocou as técnicas escalonadas dinamicamente. A Seção C.7 também é uma breve introdução ao algoritmo de Tomasulo mais complexo, abordado no Capítulo 3. Embora o algoritmo de Tomasulo possa ser abordado e entendido sem a introdução ao scoreboarding, a técnica de scoreboarding é mais simples e mais fácil de compreender.

O que é pipelining?

Pipelining é uma técnica de implementação na qual várias instruções são sobrepostas na execução; ela tira proveito do paralelismo que existe entre as ações necessárias para executar uma instrução. Hoje, o pipelining é a principal técnica de implementação usada para tornar as CPUs rápidas.

Um pipeline é como uma linha de montagem. Em uma linha de montagem de automóveis, existem muitas etapas, cada qual contribuindo com algo para a construção do carro. Cada etapa opera em paralelo com as demais, embora para carros diferentes. Em um pipeline de computador, cada etapa completa parte de uma instrução. Assim como a linha de montagem, diferentes etapas completam diferentes partes de diferentes instruções em paralelo. Cada uma dessas etapas é chamada *estágio de pipe* ou *segmento de pipe*. Os estágios são conectados um ao outro para formar uma pipe — instruções entram em uma ponta, prosseguem pelos estágios e saem na outra ponta, assim como aconteceria com os carros em uma linha de montagem.

Em uma linha de montagem de automóveis, o throughput é definido como o número de carros por hora e determinado pela frequência com que um carro completo sai da linha de montagem. Da mesma forma, o throughput de um pipeline de instruções é determinado pela frequência com que uma instrução sai do pipeline. Como os estágios de pipe são conectados, todos os estágios precisam estar prontos para prosseguir ao mesmo tempo, exatamente como ocorreria em uma linha de montagem. O tempo exigido entre acionar uma instrução e um passo no pipeline é chamado *ciclo do processador*. Como todas as etapas ocorrem ao mesmo tempo, a extensão de um ciclo do processador é determinada pelo tempo exigido para o estágio de pipe mais lento, assim como em uma linha de montagem de automóveis a etapa mais longa determinaria o tempo de avanço na fila. Em um computador, esse ciclo de processador normalmente é um ciclo de um clock (às vezes dois, e raramente mais).

O objetivo do projetista do pipeline é balancear o tamanho de cada estágio do pipeline, assim como o projetista da linha de montagem tenta balancear o tempo para cada etapa no processo. Se os estágios estiverem perfeitamente balanceados, o tempo por instrução no processador com pipeline considerando condições ideais — será igual a

$$\frac{\text{Tempo por instrução na máquina sem pipeline}}{\text{Número de estágios de pipe}}$$

Sob essas condições, o ganho de velocidade do pipelining é igual ao número de estágios de pipe, assim como uma linha de montagem com n estágios pode produzir carros, de modo ideal, n vezes mais rápido. Porém, normalmente, os estágios não serão perfeitamente balanceados; além do mais, o pipelining envolve algum overhead. Assim, o tempo por instrução no processador em pipeline não terá seu valor mínimo possível, embora possa ser próximo.

O pipelining gera uma redução no tempo médio de execução por instrução. Dependendo do que você considere como linha de base, a redução pode ser vista como a diminuição do número de ciclos de clock por instrução (CPI), como a diminuição do tempo do ciclo de clock ou como uma combinação delas. Se o ponto de partida for um processador que utiliza vários ciclos de clock por instrução, normalmente considera-se que o pipelining reduz o CPI. Essa é a visão principal que teremos. Se o ponto de partida for um processador que gasta um (longo) ciclo de clock por instrução, o pipelining diminuirá o tempo do ciclo de clock.

O pipelining é uma técnica de implementação que explora o paralelismo entre as instruções em um fluxo sequencial de instruções. Ele tem a vantagem substancial — diferentemente de algumas técnicas de ganho de velocidade (Cap. 4) — de não ser visível ao programador. Neste apêndice, primeiro veremos o conceito de pipelining usando um pipeline clássico de cinco estágios; outros capítulos investigam as técnicas de pipelining mais sofisticadas em uso nos processadores modernos. Antes de falarmos mais sobre pipelining e seu uso em um processador, precisamos de um conjunto de instruções simples, que apresentamos em seguida.

Fundamentos de um conjunto de instruções RISC

Por todo este livro, usamos uma arquitetura RISC (Reduced Instruction Set Computer) ou uma arquitetura load-store para ilustrar os conceitos básicos, embora quase todas as ideias que introduzimos neste livro se apliquem a outros processadores. Nesta seção, apresentamos o núcleo de uma arquitetura RISC típica. Neste apêndice, e em todo o livro, nossa arquitetura RISC padrão é MIPS. Em muitos lugares, os conceitos são tão parecidos que se aplicarão a qualquer RISC. As arquiteturas RISC são caracterizadas por algumas propriedades-chave que simplificam bastante sua implementação:

- Todas as operações sobre dados se aplicam aos dados nos registradores e normalmente mudam o registrador inteiro (32 ou 64 bits por registrador).
- As únicas operações que afetam a memória são operações de carregamento (load) e armazenamento (store) que movem dados da memória para um registrador ou de um registrador para a memória, respectivamente. Operações de load e store que carregam e armazenam menos do que um registrador inteiro (p. ex., um byte, 16 bits ou 32 bits) normalmente estão disponíveis.
- O número dos formatos de instrução são poucos, e todas as instruções normalmente são do mesmo tamanho.

Essas propriedades simples ocasionam simplificações drásticas na implementação do pipelining, que é o motivo desses conjuntos de instruções terem sido projetados dessa maneira.

Por consistência com o restante do texto, usamos o MIPS64, versão de 64 bits do conjunto de instruções MIPS. As instruções de 64 bits estendidas geralmente são designadas por uma letra D no início ou no final do mnemônico. Por exemplo, DADD é a versão de 64 bits de uma instrução add, enquanto LD é a versão de 64 bits de uma instrução de load.

Assim como outras arquiteturas RISC, o conjunto de instruções MIPS oferece 32 registradores, embora o registrador 0 sempre tenha o valor 0. A maioria das arquiteturas RISC, como MIPS, possui três classes de instruções (veja detalhes no Apêndice A):

1. *Instruções da ALU.* Essas instruções apanham dois registradores ou um registrador e um imediato estendido com o valor do sinal (chamadas instruções imediatas da ALU, elas possuem um offset de 16 bits no MIPS), operam sobre elas e armazenam o resultado em um terceiro registrador. As operações típicas incluem adição (`DADD`),

subtração (DSUB) e operações lógicas (como AND ou OR), que não diferenciam entre versões de 32 bits e 64 bits. As versões imediatas dessas instruções utilizam os mesmos mnemônicos com um sufixo I. No MIPS, existem os formatos sinalizado e não sinalizado das instruções aritméticas; os formatos não sinalizados, que não geram exceções por overflow — e, portanto, são iguais no modo de 32 bits e de 64 bits — possuem um U (de unsigned) no final (p. ex., DADDU, DSUBU, DADDIU).

2. *Instruções de load e store.* Essas instruções utilizam um registrador-fonte, chamado de *registrador de base*, e um campo imediato (16 bits no MIPS), chamado *offset*, como operandos. A soma — chamada *endereço efetivo* — do conteúdo do registrador de base e do offset estendido com o valor do sinal é usada como um endereço de memória. No caso de uma instrução load, um segundo operando registrador atua como um destino para os dados carregados da memória. No caso de um store, o segundo operando registrador é a origem dos dados que estão armazenados na memória. As instruções load de palavra (LD) e store de palavra (SD) carregam ou armazenam o conteúdo de registrador inteiro de 64 bits.
3. *Desvios e saltos.* Desvios são transferências condicionais do controle de execução do programa. Normalmente, existem duas maneiras de especificar a condição de desvio nas arquiteturas RISC: com um conjunto de bits de condição (às vezes chamado código de condição) ou com um conjunto limitado de comparações entre um par de registradores ou entre um registrador e zero. O MIPS utiliza o segundo. Para este apêndice, consideramos apenas as comparações de igualdade entre dois registradores. Em todas as arquiteturas RISC, o destino do desvio é obtido somando-se um offset estendido com o valor do sinal (16 bits no MIPS) ao PC atual. Os saltos incondicionais são fornecidos em muitas arquiteturas RISC, mas não os veremos neste apêndice.

Uma implementação simples de um conjunto de instruções RISC

Para entender como um conjunto de instruções RISC pode ser implementado em um padrão de pipeline, precisamos entender como ele é implementado *sem* o pipelining. Esta seção mostra uma implementação simples, em que cada instrução utiliza no máximo cinco ciclos de clock. Estenderemos essa implementação básica a uma versão em pipeline, resultando em um CPI muito menor. Nossa implementação sem pipeline não é a mais econômica ou a implementação de mais alto desempenho sem o pipelining. Em vez disso, ela foi projetada para levar naturalmente a uma implementação em pipeline. A implementação do conjunto de instruções exige a introdução de vários registradores temporários que não fazem parte da arquitetura; estes são introduzidos nesta seção para simplificar o pipelining. Nossa implementação enfocará apenas um pipeline para um subconjunto inteiro de uma arquitetura RISC, que consiste em load-store de palavra, desvio e operações com inteiros na ALU.

Cada instrução nesse subconjunto RISC pode ser implementada em, no máximo, cinco ciclos de clock. Os cinco ciclos de clock são os seguintes:

1. *Ciclo de busca de instrução* (Instruction Fetch — IF):

   Envia o contador de programa (PC — Program Counter) à memória e busca a instrução atual a partir da memória. Atualiza o PC para o próximo PC sequencial somando 4 (pois cada instrução utiliza 4 bytes) ao PC.

2. *Decodificação de instrução/ciclo de busca do registrador* (Instruction Decode — ID):

   Decodifica a instrução e lê os registradores correspondentes aos especificadores de registradores, do banco de registradores (file registers). Testa a igualdade nos registradores à medida que eles forem lidos, para um possível desvio. Estende o campo de offset da instrução com o valor do sinal, caso seja necessário. Calcula o possível

endereço de destino do desvio somando o offset estendido com o valor do sinal ao PC incrementado. Em uma implementação agressiva, que exploraremos mais adiante, o desvio pode ser completado ao final desse estágio, armazenando o endereço de destino do desvio no PC, se o teste de condição tiver um resultado verdadeiro.

A decodificação é feita em paralelo com a leitura dos registradores, o que é possível porque os especificadores de registradores ficam em um local fixo em uma arquitetura RISC. Essa técnica é conhecida como *decodificação de campo fixo*. Observe que podemos ler um registrador que não iremos usar, o que não ajuda mas também não atrapalha o desempenho. (Isso desperdiça energia para ler um registrador desnecessário, e os projetos sensíveis à potência poderão evitar isso.) Como a parte imediata de uma instrução também está em um local idêntico, o imediato com sinal estendido é calculado durante esse ciclo, caso seja necessário.

3. *Execução/ciclo de endereço efetivo* (EX):

   A ALU opera sobre os operandos preparados no ciclo anterior, realizando uma das três funções, dependendo do tipo de instrução:
   - Referência de memória: a ALU soma o registrador de base e o offset para formar o endereço efetivo.
   - Instrução da ALU Registrador-Registrador: a ALU realiza a operação especificada pelo opcode da ALU sobre os valores lidos do banco de registradores.
   - Instrução da ALU Registrador-Imediato: a ALU realiza a operação especificada pelo opcode da ALU sobre o primeiro valor lido do banco de registradores e o imediato com sinal estendido.

   Em uma arquitetura load-store, o endereço efetivo e os ciclos de execução podem ser combinados em um único ciclo de clock, pois nenhuma instrução precisa calcular simultaneamente um endereço de dados e realizar uma operação sobre os dados.

4. *Acesso à memória* (MEM):

   Se a instrução for um load, a memória faz uma leitura usando o endereço efetivo calculado no ciclo anterior. Se for um store, a memória escreve os dados do segundo registrador lido do banco de registradores usando o endereço efetivo.

5. *Ciclo de write-back* (WB):
   - Instrução da ALU Registrador-Registrador ou instrução Load:

     Escreve o resultado no banco de registradores, venha ele do sistema de memória (para um load) ou da ALU (para uma instrução da ALU).

Nessa implementação, as instruções de desvio exigem dois ciclos, as instruções de store exigem quatro ciclos e todas as outras instruções exigem cinco ciclos. Considerando uma frequência de desvio de 12% e uma frequência de store de 10%, uma distribuição de instrução típica leva a um CPI geral de 4,54. Porém, essa implementação não é ideal para conseguir o melhor desempenho nem no uso da quantidade mínima de hardware, dado o nível de desempenho; deixaremos a melhoria desse projeto como exercício para você e enfocaremos o pipelining nessa versão.

O pipeline clássico de cinco estágios para um processador RISC

Podemos usar o pipeline para a execução descrita quase sem mudanças, simplesmente iniciando uma nova instrução em cada ciclo de clock (veja por que escolhemos esse projeto!). Cada um dos ciclos de clock da seção anterior se torna um *estágio de pipe* — um ciclo no pipeline. Isso resulta no padrão de execução mostrado na Figura C.1, que é

	Número do clock								
Número da instrução	**1**	**2**	**3**	**4**	**5**	**6**	**7**	**8**	**9**
Instrução *i*	IF	ID	EX	MEM	WB				
Instrução *i* + 1		IF	ID	EX	MEM	WB			
Instrução *i* + 2			IF	ID	EX	MEM	WB		
Instrução *i* + 3				IF	ID	EX	MEM	WB	
Instrução *i* + 4					IF	ID	EX	MEM	WB

FIGURA C.1 Pipeline RISC simples.
A cada ciclo de clock, outra instrução é lida e inicia sua execução em cinco ciclos. Se uma instrução for iniciada a cada ciclo de clock, o desempenho será até cinco vezes o de um processador sem pipeline. Os nomes para os estágios no pipeline são iguais aos usados para os ciclos na implementação sem pipeline: IF = instruction fetch (busca de instrução), ID = instruction decode (decodificação de instrução), EX = execução, MEM = acesso à memória e WB = write-back.

a representação típica de uma estrutura de pipeline. Embora cada instrução utilize cinco ciclos de clock para ser concluída, durante cada ciclo de clock o hardware iniciará uma nova instrução e estará executando alguma parte das cinco instruções diferentes.

Podemos achar difícil acreditar que o pipelining seja tão simples, e não é. Nesta e nas próximas seções, tornaremos nosso pipeline RISC "real", tratando dos problemas que o pipelining introduz.

Para começar, temos que determinar o que acontece em cada ciclo de clock do processador e nos certificar de não tentarmos realizar duas operações diferentes com o mesmo recurso do datapath no mesmo ciclo de clock. Por exemplo, uma única ALU não pode ser solicitada a calcular um endereço efetivo e realizar uma operação de subtração ao mesmo tempo. Assim, temos que garantir que a sobreposição de instruções no pipeline não cause tal conflito. Felizmente, a simplicidade de um conjunto de instruções RISC torna a avaliação de recurso relativamente fácil. A Figura C.2 mostra uma versão simplificada de um datapath RISC, projetado com pipeline. Como você pode ver, as principais unidades funcionais são usadas em diferentes ciclos, por isso a sobreposição da execução de múltiplas instruções introduz relativamente poucos conflitos. Existem três observações em que esse fato se baseia.

Primeiro, usamos memórias separadas para instruções e dados, o que normalmente implementaríamos com caches de instruções e dados separadas (discutido no Cap. 2). O uso de caches separadas elimina um conflito para uma única memória, que surgiria entre a busca da instrução e o acesso à memória de dados. Observe que, se o seu processador com pipeline tiver um ciclo de clock igual ao da versão sem pipeline, o sistema de memória precisará oferecer cinco vezes a largura de banda. Essa demanda aumentada é um custo do desempenho mais alto.

Segundo, o banco de registradores é usado nos dois estágios: um para leitura em ID e um para escrita em WB. Esses usos são distintos, de modo que simplesmente mostramos o banco de registradores em dois locais. Logo, precisamos realizar duas leituras e uma escrita a cada ciclo de clock. Para lidar com duas leituras e uma escrita para o mesmo registrador (e por outro motivo, que se tornará óbvio em breve), realizamos a escrita do registrador na primeira metade do ciclo de clock e a leitura na segunda metade.

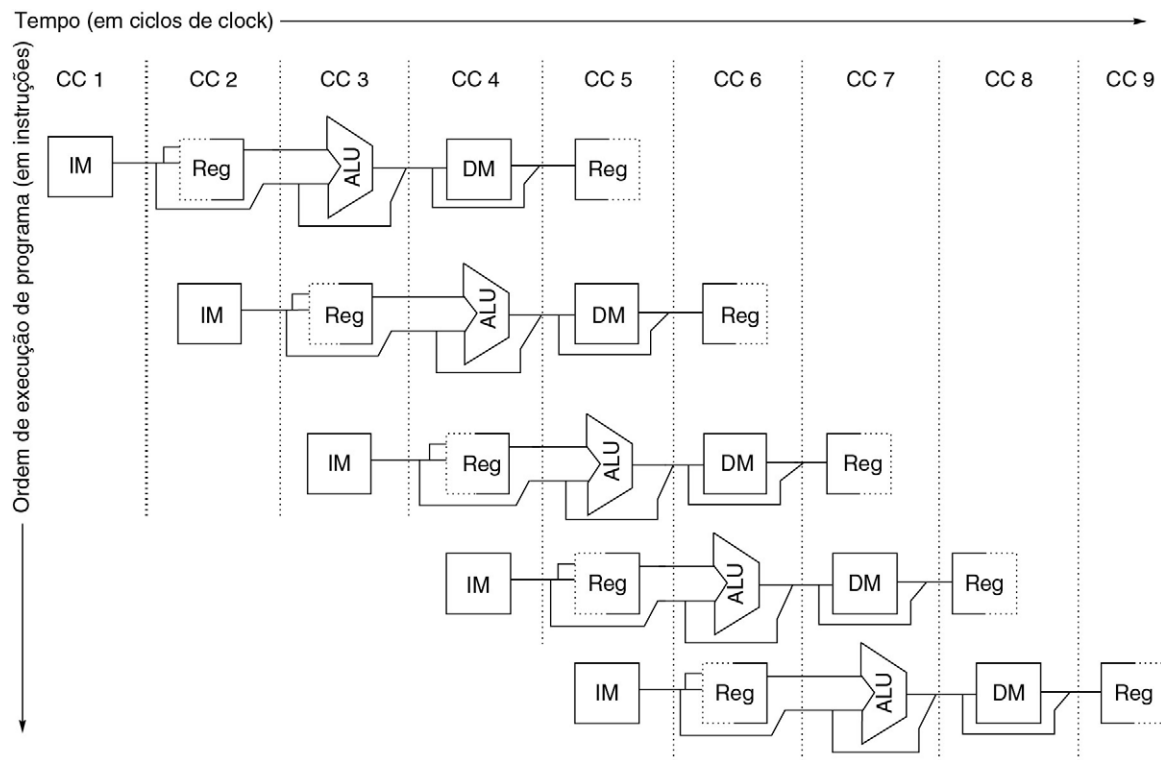



FIGURA C.2 O pipeline pode ser imaginado como uma série de datapaths deslocados no tempo.
Isso mostra a sobreposição entre as partes do datapath, com o ciclo de clock 5 (CC 5) mostrando a situação de estado fixo. Como o banco de registradores é usado como uma fonte no estágio ID e como um destino no estágio WB, ele aparece duas vezes. Mostramos que ele é lido em uma parte do estágio e escrito em outra usando uma linha sólida, à direita ou à esquerda, respectivamente, e uma linha tracejada no outro lado. A abreviação IM é usada para Instruction Memory (memória de instrução), DM para Data Memory (memória de dados) e CC para ciclo de clock.

Terceiro, a Figura C.2 não lida com o PC. Para iniciar uma nova instrução a cada clock, temos que incrementar e armazenar o PC a cada clock, e isso precisa ser feito durante o estágio de IF em preparação para a próxima instrução. Além do mais, também precisamos ter um somador para calcular o alvo do desvio em potencial durante o ID. Outro problema é que um desvio não muda o PC antes do estágio ID. Isso causa um problema, que ignoraremos por enquanto, mas de que trataremos em breve.

Embora seja fundamental garantir que as instruções no pipeline não tentem usar os recursos de hardware ao mesmo tempo, também temos que assegurar que as instruções em diferentes estágios do pipeline não interfiram umas nas outras. Essa separação é feita com a introdução de *registradores de pipeline* entre os estágios sucessivos do pipeline, de modo que, ao final de um ciclo de clock, todos os resultados de determinado estágio sejam armazenados em um registrador que é usado como entrada para o estágio seguinte no próximo ciclo de clock. A Figura C.3 mostra o pipeline desenhado com esses registradores de pipeline.

Embora muitas figuras omitam tais registradores para simplificar, eles são necessários para que o pipeline opere corretamente e precisam estar presentes. Naturalmente, registradores semelhantes seriam necessários até mesmo em um datapath multiciclos que não tivesse pipelining (pois somente os valores nos registradores são preservados entre os limites de clock). No caso de um processador com pipeline, os registradores de pipeline também desempenham o papel-chave de transportar resultados intermediários de um estágio para

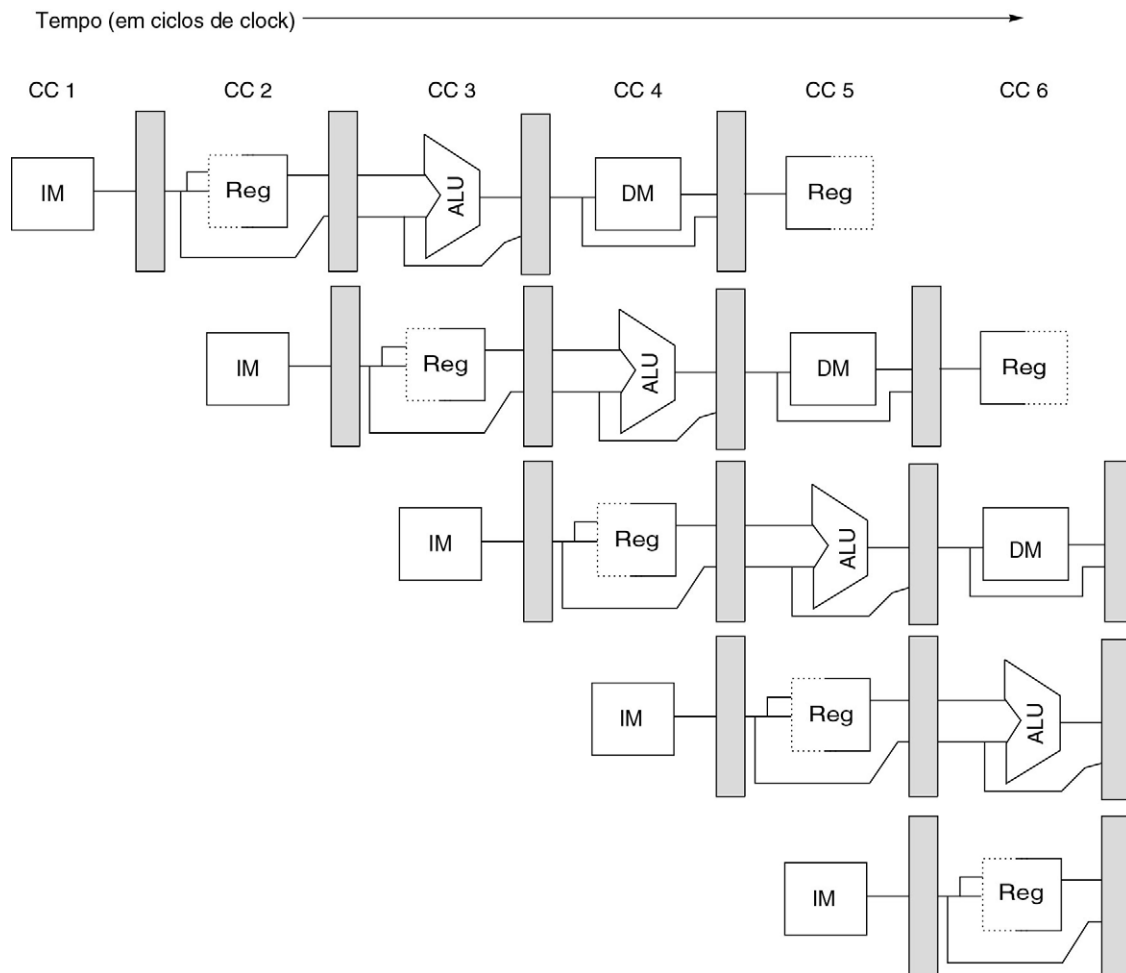



FIGURA C.3 **Pipeline mostrando os registradores de pipeline entre os sucessivos estágios de pipeline.**
Observe que os registradores impedem a interferência entre duas instruções diferentes em estágios adjacentes no pipeline. Os registradores também desempenham o papel crítico de transportar dados para determinada instrução de um estágio para o outro. A propriedade "sensível à borda" dos registradores — ou seja, os valores mudam instantaneamente em uma borda de clock — é crítica. Caso contrário, os dados de uma instrução poderiam interferir na execução de outra!

outro, onde a origem e o destino podem não ser diretamente adjacentes. Por exemplo, o valor do registrador a ser armazenado durante uma instrução de load é lido durante o ID, mas não é realmente usado antes do MEM; ele é passado por dois registradores de pipeline para atingir a memória de dados durante o estágio MEM. De modo semelhante, o resultado de uma instrução ALU é calculado durante EX, mas não é realmente armazenado antes de WB; ele chega lá pela passagem por dois registradores de pipeline. Às vezes é útil nomear os registradores de pipeline, e seguimos a convenção de nomeá-los pelos estágios de pipeline a que se conectam, de modo que os registradores são chamados IF/ID, ID/EX, EX/MEM e MEM/WB.

Questões básicas de desempenho no pipelining

O pipelining aumenta o throughput de instruções da CPU — número de instruções completadas por unidade de tempo —, mas não reduz o tempo de execução de uma instrução individual. Na verdade, em geral ele aumenta ligeiramente o tempo de execução de cada

instrução, devido ao overhead no controle do pipeline. O aumento no throughput da instrução significa que um programa roda mais rápido e possui tempo de execução total menor, embora nenhuma instrução isolada seja mais rápida!

O fato de que o tempo de execução de cada instrução não diminui coloca limites sobre a profundidade prática de um pipeline, conforme veremos na próxima seção. Além das limitações que surgem da latência do pipeline, os limites surgem do desequilíbrio entre os estágios de pipe e do overhead do pipelining. O desequilíbrio entre os estágios de pipe reduz o desempenho, pois o clock pode não executar mais rápido do que o tempo necessário para o estágio de pipeline mais lento. O overhead do pipeline surge da combinação do atraso de registrador de pipeline e do clock skew. Os registradores de pipeline aumentam o tempo de setup, que é o tempo que uma entrada de um registrador precisa estar estável antes que ocorra o sinal de clock que dispare uma escrita mais o atraso de propagação para o ciclo de clock. O clock skew, que é o atraso máximo entre a chegada do clock a dois registradores, também contribui para um limite menor no ciclo de clock. Quando o ciclo de clock é tão pequeno quanto a soma do clock skew e o overhead do latch, nenhum outro pipelining é útil, pois não existe tempo restante no ciclo para um trabalho útil. O leitor interessado deverá consultar Kunkel e Smith (1986). Como vimos no Capítulo 3, esse overhead afetou os ganhos de desempenho alcançados pelo Pentium 4 *versus* o Pentium III.

Exemplo Considere o processador sem pipeline da seção anterior. Suponha que ele tenha um ciclo de clock de 1 ns e que use quatro ciclos para as operações da ALU e desvios, e cinco ciclos para operações com a memória. Suponha também que as frequências relativas dessas operações sejam 40%, 20% e 40%, respectivamente. Suponha ainda que, devido ao clock skew e ao setup, o pipelining do processador acrescente 0,2 ns de overhead ao clock. Ignorando qualquer impacto na latência, quanto ganho de velocidade na taxa de execução da instrução obteremos com um pipeline?

Resposta O tempo médio de execução de uma instrução no processador sem pipeline é

$$\begin{aligned}\text{Tempo médio de execução da instrução} &= \text{Ciclo de clock} \times \text{CPI médio}\\ &= 1\,\text{ns} \times ((40\% + 20\%) \times 4 + 40\% \times 5)\\ &= 1\,\text{ns} \times 4{,}4\\ &= 4{,}4\,\text{ns}\end{aligned}$$

Na implementação com pipeline, o clock precisa rodar na velocidade do estágio mais lento mais o overhead, que será 1 + 0,2 ou 1,2 ns; esse é o tempo médio de execução da instrução. Assim, o ganho de velocidade advindo do pipelining é

$$\begin{aligned}\text{Ganho de velocidade do pipeline} &= \frac{\text{Tempo médio de instrução sem pipeline}}{\text{Tempo médio de instrução com pipeline}}\\ &= \frac{4{,}4\,\text{ns}}{1{,}2\,\text{ns}} = 3{,}7\end{aligned}$$

O overhead de 0,2 ns basicamente estabelece um limite sobre a eficácia do pipelining. Se o overhead não for afetado pelas mudanças no ciclo de clock, a lei de Amdahl nos diz que o overhead limita o ganho de velocidade.

Esse pipeline RISC simples funcionaria muito bem para instruções de inteiros se cada instrução fosse independente de outra no pipeline. Na realidade, as instruções no pipeline podem depender umas das outras; esse é o assunto da próxima seção.

C.2 O PRINCIPAL OBSTÁCULO DO PIPELINING — HAZARDS DO PIPELINE

Existem situações, chamadas *hazards*, que impedem que a próxima instrução no fluxo de instruções seja executada durante seu ciclo de clock designado. Os hazards reduzem o desempenho do ganho de velocidade ideal obtido pelo pipelining. Existem três classes de hazards:

1. *Hazards estruturais* surgem de conflitos de recursos quando o hardware não pode aceitar todas as combinações possíveis de instruções simultaneamente em execução sobreposta.
2. *Hazards de dados* surgem quando uma instrução depende dos resultados de uma instrução anterior de uma maneira que é exposta pela sobreposição de instruções no pipeline.
3. *Hazards de controle* surgem no pipelining de desvios e outras instruções que mudam o PC.

Os hazards nos pipelines podem tornar necessário um *stall* no pipeline. Evitar um hazard normalmente exige que algumas instruções no pipeline tenham permissão para prosseguir, enquanto outras são adiadas. Para os pipelines que discutimos neste apêndice, quando uma instrução é adiada, todas as instruções despachadas *depois* da instrução adiada — e, portanto, não tão longe no pipeline — também são adiadas. As instruções despachadas *antes* da instrução adiada — e, portanto, mais distantes no pipeline — precisam continuar, pois de outra forma o hazard nunca será limpo. Como resultado, nenhuma instrução nova é apanhada durante o stall. Nesta seção veremos diversos exemplos de como os stalls do pipeline operam. Não se preocupe, eles não são tão complexos quanto parecem!

Desempenho dos pipelines com stalls

Um stall faz com que o desempenho do pipeline fique menor do que o desempenho ideal. Vejamos uma equação simples para descobrir o ganho de velocidade real advindo do pipelining, começando com a fórmula da seção anterior:

$$\begin{aligned}\text{Ganho de velocidade do pipeline} &= \frac{\text{Tempo médio de instrução sem pipeline}}{\text{Tempo médio de instrução com pipeline}}\\ &= \frac{\text{CPI sem pipeline} \times \text{Ciclo de clock sem pipeline}}{\text{CPI com pipeline} \times \text{Ciclo de clock com pipeline}}\\ &= \frac{\text{CPI sem pipeline} \times \text{Ciclo de clock sem pipeline}}{\text{CPI com pipeline} \times \text{Ciclo de clock com pipeline}}\end{aligned}$$

O pipelining pode ser considerado como diminuindo o CPI ou o tempo de ciclo de clock. Como é tradicional usar o CPI para comparar os pipelines, vamos começar com essa suposição. O CPI ideal em um processador com pipeline é quase sempre 1. Logo, podemos calcular o CPI com pipeline:

$$\begin{aligned}\text{CPI com pipeline} &= \text{CPI ideal} + \text{Ciclos de clock de stall do pipeline por instrução}\\ &= 1 + \text{Ciclos de clock de stall do pipeline por instrução}\end{aligned}$$

Se ignorarmos o overhead do tempo de ciclo do pipelining e assumirmos que os estágios estão perfeitamente balanceados, o tempo de ciclo dos dois processadores pode ser igual, levando a

$$\text{Ganho de velocidade do pipeline} = \frac{\text{CPI sem pipeline}}{1 + \text{Ciclos de clock de stall do pipeline por instrução}}$$

Um caso simples importante é aquele em que todas as instruções utilizam o mesmo número de ciclos, que também precisa ser igual ao número de estágios de pipeline (também chamado *profundidade do pipeline*). Nesse caso, o CPI sem pipeline é igual à profundidade do pipeline, levando a

$$\text{Ganho de velocidade do pipeline} = \frac{\text{Profundidade do pipeline}}{1 + \text{Ciclos de clock de stall do pipeline por instrução}}$$

Se não houver stalls no pipeline, isso levará ao resultado intuitivo de que o pipelining pode melhorar o desempenho pela profundidade do pipeline.

Como alternativa, se pensarmos no pipelining como melhorando o tempo de ciclo de clock, poderemos considerar que o CPI do processador sem pipeline, além do processador com pipeline, é 1. Isso nos leva a

$$\begin{aligned}\text{Ganho de velocidade do pipeline} &= \frac{\text{CPI sem pipeline}}{\begin{array}{c}1 + \text{Ciclos de clock de stall}\\ \text{do pipeline por instrução}\end{array}} \times \frac{\text{Ciclo de clock sem pipeline}}{\text{Ciclo de clock com pipeline}}\\ &= \frac{1}{\begin{array}{c}1 + \text{Ciclos de clock de stall}\\ \text{do pipeline por instrução}\end{array}} \times \text{Profundidade do pipeline}\end{aligned}$$

Em casos nos quais os estágios de pipe são perfeitamente balanceados e não existe overhead, o ciclo de clock no processador em pipeline é menor do que o ciclo de clock do processador sem pipeline por um fator igual à profundidade do pipeline:

$$\begin{aligned}\text{Ciclo de clock com pipeline} &= \frac{\text{Ciclo de clock sem pipeline}}{\text{Profundidade do pipeline}}\\ \text{Profundidade do pipeline} &= \frac{\text{Ciclo de clock sem pipeline}}{\text{Ciclo de clock com pipeline}}\end{aligned}$$

Isso nos leva ao seguinte:

$$\begin{aligned}\text{Ganho de velocidade do pipeline} &= \frac{1}{\begin{array}{c}1 + \text{Ciclos de clock de stall}\\ \text{do pipeline por instrução}\end{array}} \times \frac{\text{Ciclo de clock sem pipeline}}{\text{Ciclo de clock com pipeline}}\\ &= \frac{1}{\begin{array}{c}1 + \text{Ciclos de clock de stall}\\ \text{do pipeline por instrução}\end{array}} \times \text{Profundidade do pipeline}\end{aligned}$$

Assim, se não houver stalls, o ganho de velocidade será igual ao número de estágios do pipeline, correspondendo à nossa intuição para o caso ideal.

Hazards estruturais

Quando um processador está em pipeline, a execução superposta das instruções requer o pipelining das unidades funcionais e a duplicação dos recursos para permitir todas as combinações de instruções possíveis no pipeline. Se alguma combinação de instruções não puder ser acomodada devido a conflitos de recurso, o processador será considerado como tendo um *hazard estrutural*.

Os casos mais comuns de hazards estruturais surgem quando alguma unidade funcional não está totalmente em pipeline. Então, uma sequência de instruções usando essa unidade sem pipeline não pode prosseguir na taxa de uma por ciclo de clock. Outra forma comum como os hazards estruturais aparecem é quando algum recurso não foi duplicado o suficiente

para permitir que todas as combinações de instruções no pipeline sejam executadas. Por exemplo, um processador pode ter apenas uma porta de escrita no banco de registradores, mas, sob certas circunstâncias, o pipeline poderia querer realizar duas escritas em um ciclo de clock. Isso geraria um hazard estrutural.

Quando uma sequência de instruções encontra esse hazard, o pipeline adia uma das instruções até que a unidade solicitada esteja disponível. Esses stalls aumentarão o CPI do seu valor ideal normal de 1.

Alguns processadores compartilharam um pipeline de única memória para dados e instruções. Como resultado, quando uma instrução contém uma referência de memória de dados, ela entra em conflito com a referência de instrução para a próxima instrução, como mostra a Figura C.4. Para resolver esse hazard, adiamos o pipeline por um ciclo de clock quando ocorre o acesso à memória de dados. Um stall normalmente é chamado *bolha de pipeline* ou apenas *bolha*, pois flutua pelo pipeline ocupando espaço, porém sem transportar trabalho útil. Veremos outro tipo de stall quando falarmos sobre os hazards de dados.

Os projetistas normalmente indicam o comportamento do stall usando um diagrama simples apenas com os nomes de estágio de pipe, como na Figura C.5. A forma da Figura C.5 mostra o stall indicando o ciclo quando não ocorre uma ação e simplesmente deslocando a instrução 3 para a direita (o que atrasa o início e o fim de sua execução em um ciclo). O efeito da bolha de pipeline é realmente ocupar os recursos para o slot dessa instrução enquanto ela trafega pelo pipeline.

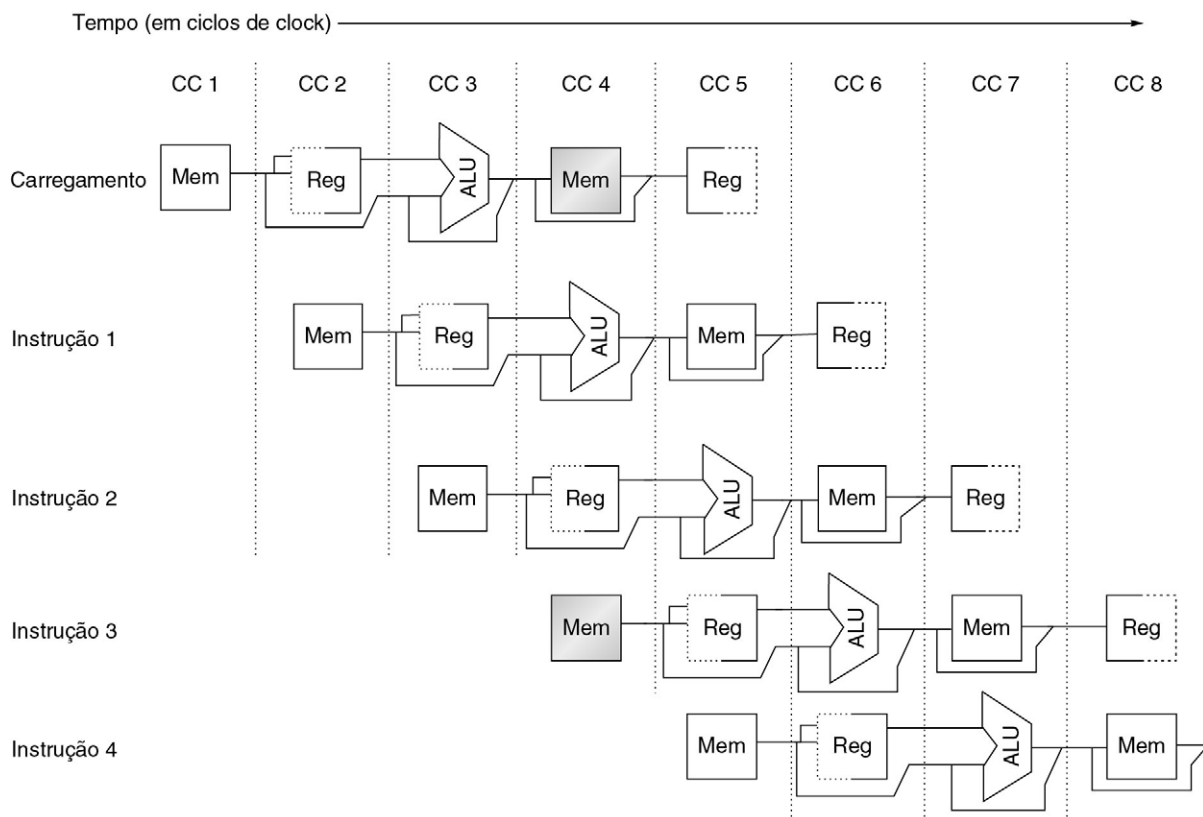



FIGURA C.4 **Um processador com apenas uma porta de memória gerará conflito sempre que houver uma referência de memória.** Neste exemplo, a instrução load utiliza a memória para um acesso aos dados ao mesmo tempo em que a instrução 3 deseja buscar uma instrução da memória.

	Número do ciclo de clock									
Instrução	**1**	**2**	**3**	**4**	**5**	**6**	**7**	**8**	**9**	**10**
Instrução load	IF	ID	EX	MEM	WB					
Instrução *i* + 1		IF	ID	EX	MEM	WB				
Instrução *i* + 2			IF	ID	EX	MEM	WB			
Instrução *i* + 3				stall	IF	ID	EX	MEM	WB	
Instrução *i* + 4						IF	ID	EX	MEM	WB
Instrução *i* + 5							IF	ID	EX	MEM
Instrução *i* + 6								IF	ID	EX

FIGURA C.5 Um pipeline com stall para um hazard estrutural — um load com uma porta de memória.
Como vemos aqui, a instrução load efetivamente rouba um ciclo de busca de instrução, causando o stall do pipeline — nenhuma instrução é iniciada no ciclo de clock 4 (que normalmente iniciaria a instrução *i* + 3). Como a instrução sendo buscada é adiada, todas as outras instruções no pipeline antes da instrução adiada podem prosseguir normalmente. O ciclo de stall continuará a passar pelo pipeline, de modo que nenhuma instrução termina no ciclo de clock 8. Às vezes, esses diagramas de pipeline são desenhados com o stall ocupando uma linha horizontal inteira e a instrução 3 sendo movida para a próxima linha; de qualquer forma, o efeito é o mesmo, pois a instrução *i* + 3 não inicia a execução até o ciclo 5. Usamos o formato anterior, pois gasta menos espaço na figura. Observe que esta figura considera que as instruções *i* + 1 e *i* + 2 não são referências à memória.

Exemplo Vejamos quanto poderia custar o hazard estrutural do load. Suponha que as referências de dados constituam 40% do total e que o CPI ideal do processador com pipeline, ignorando o hazard estrutural, seja 1. Suponha também que o processador com o hazard estrutural tenha uma taxa de clock que é 1,05 vez mais alta do que a taxa de clock do processador sem o hazard. Desconsiderando quaisquer outras perdas de desempenho, qual é mais rápida: o pipeline com ou sem hazard estrutural? Quão mais rápida?

Resposta Existem várias maneiras de resolver esse problema. Talvez a mais simples seja calcular o tempo médio da instrução nos dois processadores:

$$\text{Tempo médio da instrução} = \text{CPI} \times \text{Tempo do ciclo de clock}$$

Por não ter stalls, o tempo médio da instrução para o processador ideal é simplesmente o Tempo do ciclo de clock$_{\text{ideal}}$. O tempo médio da instrução para o processador com o hazard estrutural é:

$$\begin{aligned}\text{Tempo médio da instrução} &= \text{CPI} \times \text{Tempo do ciclo de clock}\\ &= (1+0{,}4\times 1)\times \frac{\text{Tempo do ciclo de clock}_{\text{ideal}}}{1{,}05}\\ &= 1{,}3 \times \text{Tempo do ciclo de clock}_{\text{ideal}}\end{aligned}$$

Obviamente, o processador sem hazard estrutural é mais rápido; podemos usar a razão dos tempos médios da instrução para concluir que o processador sem hazard é 1,3 vez mais rápido.

Como alternativa a esse hazard estrutural, o projetista poderia oferecer um acesso separado à memória para instruções, seja dividindo a cache em caches separadas para instruções e dados, seja usando um conjunto de buffers, normalmente chamados buffers de instruções, para manter instruções. O Capítulo 5 explica as ideias tanto sobre a cache dividida quanto sobre o buffer de instruções.

Se todos os outros fatores forem iguais, um processador sem hazards estruturais sempre terá um CPI inferior. Por que, então, um projetista permitiria hazards estruturais? O motivo principal é reduzir o custo da unidade, pois o pipelining de todas as unidades funcionais, ou sua duplicação, pode ser muito dispendioso. Por exemplo, os processadores que admitem o acesso a um cache de instruções e a um cache de dados a cada ciclo (para impedir o hazard estrutural do exemplo anterior) exigem o dobro da largura de banda de memória total e normalmente possuem largura de banda mais alta nos pinos. De modo semelhante, um multiplicador de ponto flutuante totalmente pipelining consome muitas portas. Se o hazard estrutural for raro, pode não valer a pena o custo de evitá-lo.

Hazards de dados

Um efeito importante do pipelining é alterar os tempos relativos das instruções sobrepondo sua execução. Essa sobreposição introduz hazards de dados e controle. Os hazards de dados ocorrem quando o pipeline muda a ordem de acessos de leitura/escrita para os operandos, de modo que a ordem difere da ordem vista pela execução sequencial das instruções em um processador em não pipeline. Considere a execução em pipeline destas instruções:

```
DADD    R1,R2,R3
DSUB    R4,R1,R5
AND     R6,R1,R7
OR      R8,R1,R9
XOR     R10,R1,R11
```

Todas as instruções após o DADD utilizam o resultado dessa instrução. Como vemos na Figura C.6, a instrução DADD escreve o valor de R1 no estágio de pipe WB, mas a instrução DSUB lê o valor durante seu estágio ID. Esse problema é chamado *hazard de dados*. A menos que sejam tomadas precauções para evitar isso, a instrução DSUB lerá o valor errado e tentará usá-lo. De fato, o valor usado pela instrução DSUB nem sequer é determinístico: embora possamos achar lógico considerar que DSUB sempre use o valor de R1 que foi atribuído por uma instrução antes do DADD, isso nem sempre acontece. Se uma interrupção ocorrer entre as instruções DADD e DSUB, o estágio WB do DADD terminará e o valor de R1 nesse ponto será o resultado do DADD. Esse comportamento imprevisível obviamente é inaceitável.

A instrução AND também é afetada por esse hazard. Como podemos ver pela Figura C.6, a escrita de R1 não se completa até o final do ciclo de clock 5. Assim, a instrução AND que lê os registradores durante o ciclo de clock 4 receberá os resultados errados.

A instrução XOR opera corretamente porque sua leitura de registrador ocorre no ciclo de clock 6, depois da escrita do registrador. A instrução OR também opera sem incorrer em um hazard, pois realizamos as leituras do banco de registradores na segunda metade do ciclo e as escritas na primeira metade.

A próxima subseção discutirá uma técnica para eliminar os stalls para o hazard envolvendo as instruções DSUB e AND.

Minimizando os stalls de hazard de dados pelo adiantamento

O problema imposto na Figura C.6 pode ser resolvido com uma técnica de hardware simples, chamada *adiantamento* (também chamado *bypassing* e, às vezes, *curto-circuito*). A ideia principal no adiantamento é de que o resultado não é realmente necessário pelo DSUB

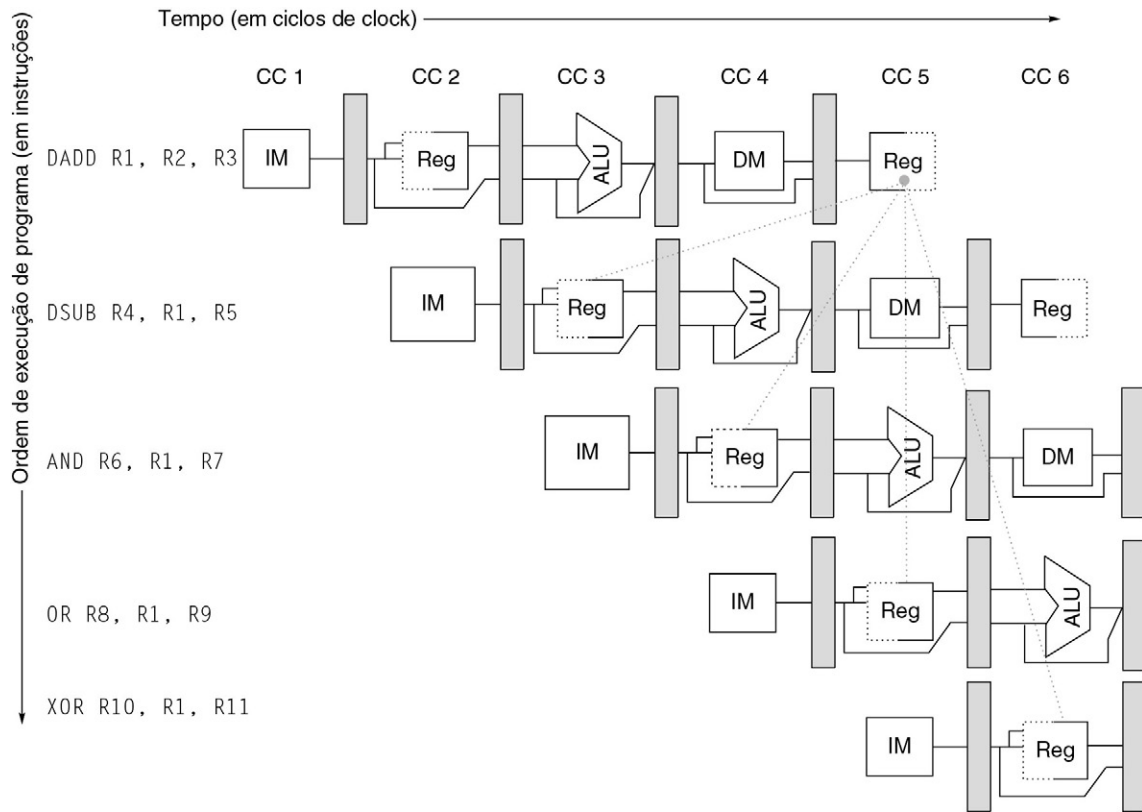



FIGURA C.6 O uso do resultado da instrução DADD nas instruções seguintes causa um hazard, pois o registrador não é escrito antes que essas instruções o leiam.

antes que o DADD realmente o produza. Se o resultado puder ser movido do registrador do pipeline onde o DADD o armazena para onde o DSUB precisa dele, então a necessidade de um stall pode ser evitada. Usando essa observação, o adiantamento funciona da seguinte maneira:

1. O resultado da ALU dos registradores de pipeline EX/MEM e MEM/WB é sempre alimentado de volta às entradas da ALU.
2. Se o hardware de adiantamento detectar que a operação anterior da ALU escreverá no registrador correspondente à fonte para a operação atual da ALU, a lógica de controle seleciona o resultado adiantado como entrada da ALU, em vez do valor lido do banco de registradores.

Observe que, com o adiantamento, se o DSUB for adiado, o DADD será completado e o bypass não será ativado. Esse relacionamento também é verdadeiro para o caso de uma interrupção entre as duas instruções.

Como mostra o exemplo na Figura C.6, precisamos encaminhar os resultados não apenas da instrução imediatamente anterior, mas possivelmente de uma instrução que se iniciou dois ciclos antes. A Figura C.7 mostra nosso exemplo com os caminhos de bypass incluídos e destacando os tempos de leitura e escrita de registrador. Essa sequência de código pode ser executada sem stalls.

O adiantamento pode ser generalizado para incluir a passagem de um resultado diretamente para a unidade funcional que o exige: um resultado é encaminhado do registrador

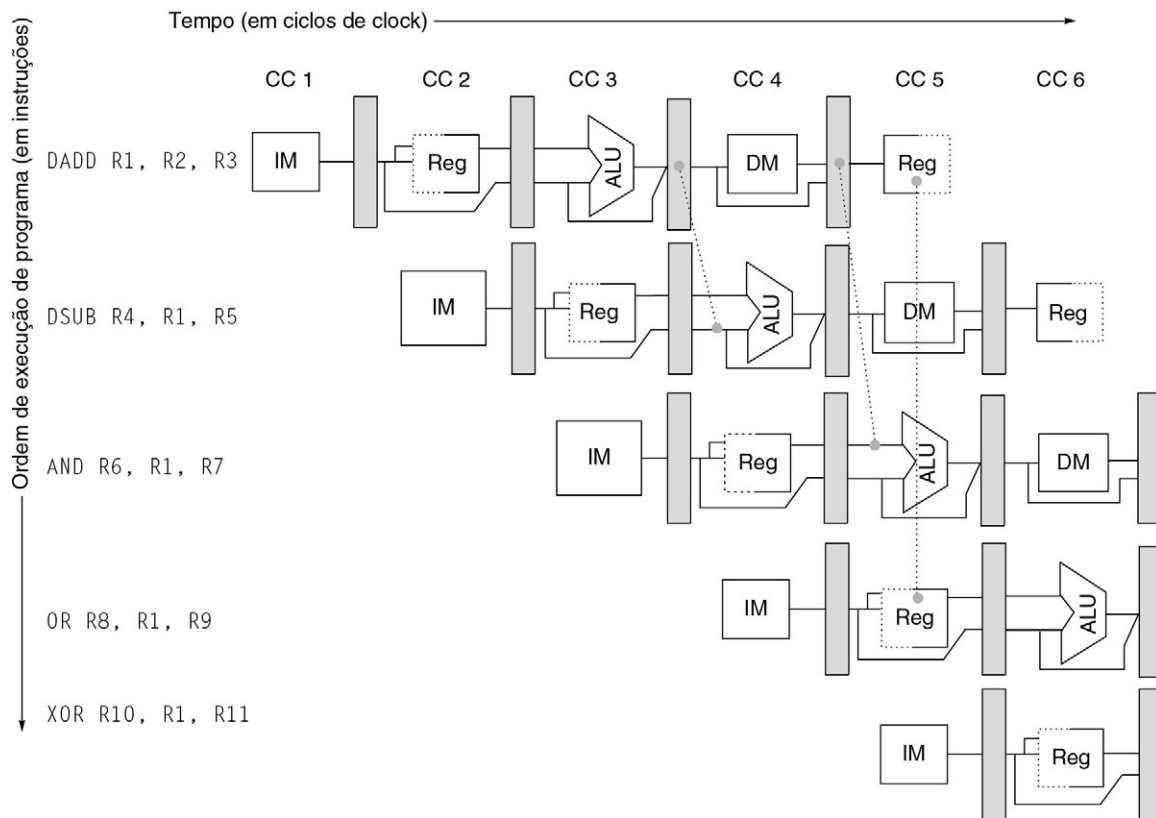



FIGURA C.7 **Um conjunto de instruções que depende do resultado do DADD usa vias de adiantamento para evitar o hazard de dados.** As entradas para as instruções DSUB e AND adiantam os valores calculados, dos registradores de pipeline para a primeira entrada da ALU. O OR recebe seu resultado no banco de registradores, que é facilmente obtido pela leitura dos registradores na segunda metade do ciclo e pela escrita na primeira metade, conforme indicam as linhas tracejadas nos registradores. Observe que o resultado adiantado pode ir para qualquer entrada da ALU; na verdade, as duas entradas da ALU poderiam usar as entradas adiantadas do mesmo registrador de pipeline ou de diferentes registradores de pipeline. Isso ocorreria, por exemplo, se a instrução AND fosse AND R6, R1, R4.

de pipeline correspondente à saída de uma unidade para a entrada de outro, e não simplesmente do resultado de uma unidade para a entrada da mesma unidade. Considere, por exemplo, a sequência a seguir:

```
DADD    R1,R2,R3
LD      R4,0(R1)
SD      R4,12(R1)
```

Para evitar um stall nessa sequência, você precisaria adiantar os valores da saída da ALU e da saída da unidade de memória dos registradores de pipeline para a ALU e para as entradas da memória de dados. A Figura C.8 mostra todas as vias de adiantamento para este exemplo.

Hazards de dados que exigem stalls

Infelizmente, nem todos os hazards de dados em potencial podem ser tratados pelo bypassing. Considere esta sequência de instruções:

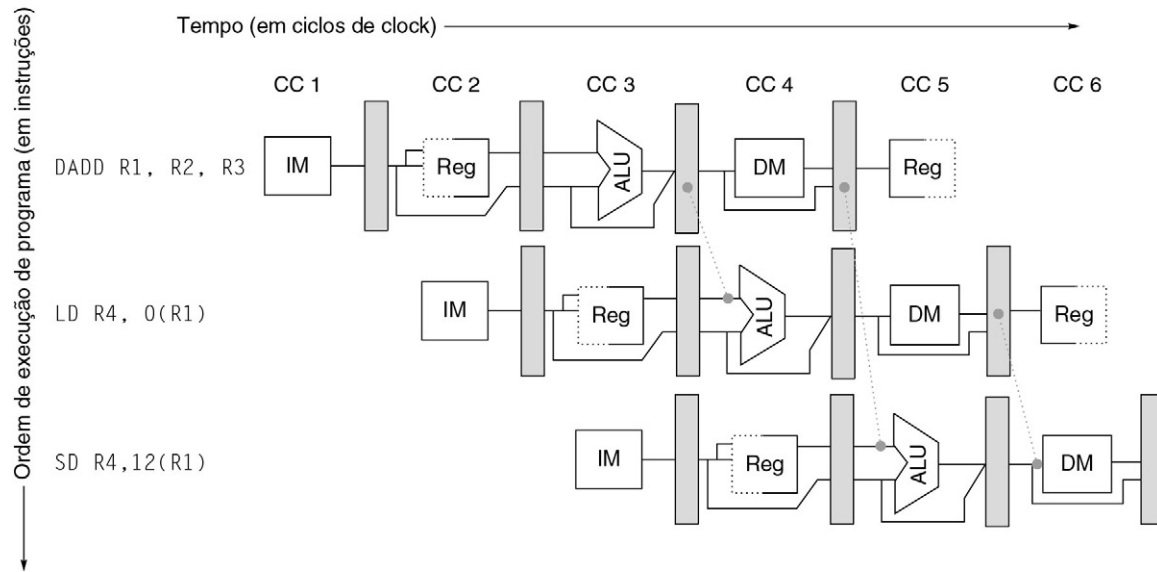



FIGURA C.8 Adiantamento do operando exigido pelos loads durante MEM.
O resultado do load é adiantado da saída da memória para a entrada da memória, a fim de ser encaminhada. Além disso, a saída da ALU é adiantada para a entrada da ALU para o cálculo de endereço do load e do store (isso não é diferente de adiantar para outra operação da ALU). Se o load dependesse de uma operação da ALU imediatamente anterior (não mostrada aqui), o resultado precisaria ser adiantado para impedir um stall.

```
LD      R1,0(R2)
DSUB    R4,R1,R5
AND     R6,R1,R7
OR      R8,R1,R9
```

O datapath com pipeline, com os caminhos de bypass para esse exemplo aparece na Figura C.9. Esse caso é diferente da situação com as operações da ALU de ponta a ponta. A instrução LD não tem os dados antes do final do ciclo de clock 4 (seu ciclo MEM), enquanto a instrução DSUB precisa ter os dados no início desse ciclo de clock. Assim, o hazard de dados do uso do resultado de uma instrução de load não pode ser eliminado completamente simplesmente com o hardware. Como mostra a Figura C.9, essa via de adiantamento teria que operar ao contrário no tempo — uma capacidade ainda não disponível aos projetistas de computador! *Podemos* adiantar o resultado imediatamente para a ALU a partir dos registradores de pipeline para uso na operação AND, que inicia dois ciclos de clock após o load. De modo semelhante, a instrução OR não tem problema, pois recebe o valor através do banco de registradores. Para a instrução DSUB, o resultado encaminhado chega muito tarde — no final de um ciclo de clock, quando é necessário no início.

A instrução load possui um atraso ou latência que não pode ser eliminado apenas pelo adiantamento. Em vez disso, precisamos acrescentar um hardware, chamado *interlock de pipeline*, para preservar a execução-padrão correta. Em geral, um interlock de pipeline detecta um hazard e atrasa o pipeline até que o hazard seja resolvido. Nesse caso, o interlock atrasa o pipeline, começando com a instrução que deseja usar os dados até a instrução de origem que os produz. Esse interlock de pipeline introduz um stall ou bolha, como acontecia para o hazard estrutural. O CPI para a instrução atrasada aumenta pela extensão do stall (um ciclo de clock nesse caso).

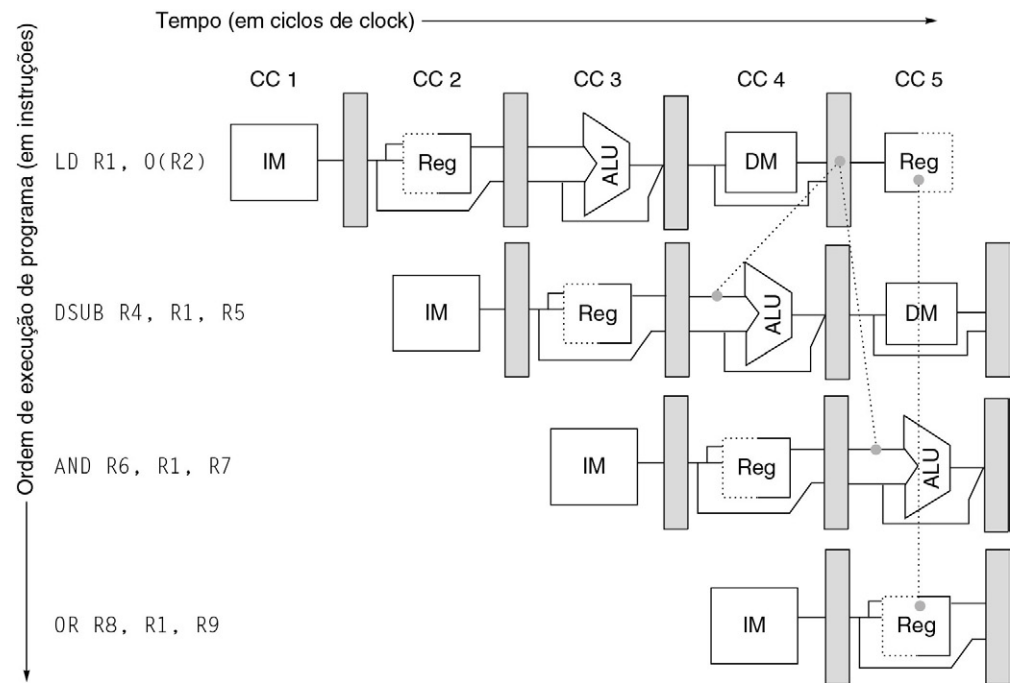



FIGURA C.9 A instrução load pode passar seus resultados para as instruções AND e OR, mas não para o DSUB, pois isso significaria encaminhar o resultado em "tempo negativo".

A Figura C.10 mostra o pipeline antes e depois do stall, usando os nomes dos estágios do pipeline. Como o stall faz com que as instruções começando com o DSUB se movam um ciclo adiante no tempo, o adiantamento para a instrução AND agora passa pelo banco de registradores e nenhum adiantamento é necessário para a instrução OR. A inserção da bolha faz com que o número de ciclos para terminar essa sequência aumente em um. Nenhuma instrução é iniciada durante o ciclo de clock 4 (e nenhuma termina durante o ciclo 6).

LD	R1,0(R2)	IF	ID	EX	MEM	WB				
DSUB	R4,R1,R5		IF	ID	EX	MEM	WB			
AND	R6,R1,R7			IF	ID	EX	MEM	WB		
OR	R8,R1,R9				IF	ID	EX	MEM	WB	

LD	R1,0(R2)	IF	ID	EX	MEM	WB				
DSUB	R4,R1,R5		IF	ID	Stall	EX	MEM	WB		
AND	R6,R1,R7			IF	Stall	ID	EX	MEM	WB	
OR	R8,R1,R9				Stall	IF	ID	EX	MEM	WB

FIGURA C.10 Na metade superior, podemos ver por que um stall é necessário: o ciclo MEM do load produz um valor que é necessário no ciclo EX do DSUB, que ocorre ao mesmo tempo.
Esse problema é solucionado pela inserção de um stall, como mostra a metade inferior.

Hazards de desvio condicionais

Os *hazards de controle* podem causar uma perda de desempenho maior para o nosso pipeline MIPS do que os hazards de dados. Quando um desvio é executado, ele pode ou não alterar o PC para algo diferente do seu valor atual mais 4. Lembre-se de que, se um desvio mudar o PC para o seu endereço de destino, ele será um desvio *tomado*; se ele passar direto, será um desvio *não tomado*. Se a instrução *i* for um desvio tomado, o PC normalmente não será alterado antes do final do ID depois do término do cálculo e comparação de endereço.

A Figura C.11 mostra que o modo mais simples de lidar com os desvios é refazer a busca da instrução após um desvio, uma vez detectado o desvio durante ID (quando as instruções são decodificadas). O primeiro ciclo IF é basicamente um stall, pois nunca realiza um trabalho útil. Você pode ter notado que, se o desvio não for tomado, a repetição do estágio IF será desnecessária, pois a instrução correta foi realmente buscada. Em breve, desenvolveremos vários esquemas para tirar proveito desse fato.

Instrução de desvio	IF	ID	EX	MEM	WB		
Sucessor do desvio		IF	IF	ID	EX	MEM	WB
Sucessor do desvio + 1				IF	ID	EX	MEM
Sucessor do desvio + 2					IF	ID	EX

FIGURA C.11 Um desvio causa um stall de um ciclo no pipeline de cinco estágios.
A instrução após o desvio é buscada, mas a instrução é ignorada e a busca é reiniciada quando o alvo do desvio é conhecido. Provavelmente, é óbvio que, se o desvio não for tomado, o segundo IF para o sucessor do desvio é redundante. Isso será resolvido em breve.

Um ciclo de stall para cada desvio gerará uma perda de desempenho de 10-30%, dependendo da frequência do desvio, de modo que examinaremos algumas técnicas para lidar com essa perda.

Reduzindo as penalidades de desvio do pipeline

Existem muitos métodos para lidar com os stalls do pipeline causados por atraso de desvio condicional; nesta subseção, discutiremos quatro esquemas simples em tempo de compilação. Nesses esquemas, as ações para um desvio são estáticas — elas são fixadas para cada desvio durante a execução inteira. O software pode tentar minimizar a penalidade do desvio usando o conhecimento do esquema de hardware e do comportamento do desvio. Os Capítulos 2 e 3 examinam técnicas mais poderosas de hardware e software para as previsões de desvio estática e dinâmica.

O esquema mais simples para lidar com os desvios é *congelar* (*freeze*) ou *esvaziar* (*flush*) o pipeline, mantendo ou excluindo quaisquer instruções após o desvio até que o destino desse desvio seja conhecido. A atratividade dessa solução está principalmente na sua simplicidade, tanto para o hardware quanto para o software. Essa é a solução usada anteriormente no pipeline mostrada na Figura C.11. Nesse caso, a penalidade do desvio é fixa e não pode ser reduzida pelo software.

Um esquema de maior desempenho e apenas ligeiramente mais complexo é tratar cada desvio como não tomado, simplesmente permitindo que o hardware continue como se o desvio não fosse executado. Aqui, deve-se ter o cuidado de não mudar o estado do processador antes que o resultado do desvio seja definitivamente conhecido. A complexidade desse esquema surge de se ter de saber quando o estado poderia ser mudado por uma instrução e como “reverter” tal mudança.

Em um pipeline simples de cinco estágios, esse esquema de *previsto não tomado* (*predicted-not-taken* ou *predicted-untaked*) é implementado continuando-se a buscar instruções como se o desvio fosse uma instrução normal. O pipeline aparece como se nada fora do comum estivesse acontecendo. Porém, se o desvio for tomado, precisamos transformar a instrução buscada em uma no-op e reiniciar a busca no endereço de destino. A Figura C.12 mostra as duas situações.

Instrução de desvio não tomado	IF	ID	EX	MEM	WB					
Instrução *i* + 1		IF	ID	EX	MEM	WB				
Instrução *i* + 2			IF	ID	EX	MEM	WB			
Instrução *i* + 3				IF	ID	EX	MEM	WB		
Instrução *i* + 4					IF	ID	EX	MEM	WB	

Instrução de desvio tomado	IF	ID	EX	MEM	WB				
Instrução *i* + 1		IF	**ocioso**	**ocioso**	**ocioso**	**ocioso**			
Alvo do desvio			IF	ID	EX	MEM	WB		
Alvo do desvio + 1				IF	ID	EX	MEM	WB	
Alvo do desvio + 2					IF	ID	EX	MEM	WB

FIGURA C.12 O esquema previsto não tomado e a sequência de pipeline quando o desvio não é tomado (em cima) e é tomado (embaixo). Quando o desvio não é tomado, determinado durante ID, buscamos o fall-through e simplesmente continuamos. Se o desvio for tomado durante ID, reiniciamos a busca no destino do desvio. Isso faz com que todas as instruções após o desvio atrasem por um ciclo de clock.

Um esquema alternativo é tratar cada desvio como tomado. Assim que o desvio é decodificado e o endereço de destino é calculado, assumimos que o desvio foi tomado e começamos a buscar e executar no destino. Como em nosso pipeline de cinco estágios não sabemos o endereço de destino antes de saber o resultado do desvio, não existe vantagem nessa técnica para esse pipeline. Em alguns processadores — especialmente aqueles com códigos de condições definidos implicitamente ou condições de desvio mais poderosas (e, portanto, mais lentas), o destino do desvio é conhecido antes do resultado do desvio, e um esquema previsto-tomado poderia fazer sentido. Em um esquema previsto-tomado ou previsto não tomado, o compilador pode melhorar o desempenho organizando o código de modo que o caminho mais frequente corresponda à escolha do hardware. Nosso quarto esquema oferece mais oportunidades para o compilador melhorar o desempenho.

Um quarto esquema em uso em alguns processadores é chamado *desvio adiado*. Essa técnica foi muito usada nos primeiros processadores RISC e funciona razoavelmente bem no pipeline de cinco estágios. Em um desvio adiado, o ciclo de execução com um atraso de desvio condicional de um é

```
instrução de desvio
sucessor na sequência1
alvo do desvio se for tomado
```

O sucessor em sequência está no *slot de atraso de desvio condicional* (*branch delayed slot*). Essa instrução é executada independentemente de o desvio ter sido tomado ou não. O

Instrução de desvio não tomado	IF	ID	EX	MEM	WB				
Instrução de atraso de desvio condicional (*i* + 1)		IF	ID	EX	MEM	WB			
Instrução *i* + 2			IF	ID	EX	MEM	WB		
Instrução *i* + 3				IF	ID	EX	MEM	WB	
Instrução *i* + 4					IF	ID	EX	MEM	WB

Instrução de desvio tomado	IF	ID	EX	MEM	WB				
Instrução de atraso de desvio condicional (*i* + 1)		IF	ID	EX	MEM	WB			
Alvo do desvio			IF	ID	EX	MEM	WB		
Alvo do desvio + 1				IF	ID	EX	MEM	WB	
Alvo do desvio + 2					IF	ID	EX	MEM	WB

FIGURA C.13 **O comportamento de um desvio adiado é o mesmo, não importa se o desvio é tomado ou não.** As instruções no slot de atraso (existe apenas um slot de atraso para o MIPS) são executadas. Se o desvio não for tomado, a execução continua com a instrução após a instrução de atraso de desvio condicional; se o desvio for tomado, a execução continua no destino do desvio. Quando a instrução no slot de atraso de desvio condicional também é um desvio, o significado é incerto: se o desvio não for tomado, o que deverá acontecer ao desvio no slot de atraso de desvio condicional? Devido a essa confusão, as arquiteturas com desvios de atraso normalmente não permitem a colocação de um desvio no slot de atraso.

comportamento do pipeline de cinco estágios com um atraso de desvio condicional aparece na Figura C.13. Embora seja possível ter um atraso de desvio condicional maior do que 1, na prática quase todos os processadores com desvio adiado possuem um único atraso de instrução; outras técnicas são usadas se o pipeline tiver uma penalidade de desvio em potencial maior.

A tarefa do compilador é tornar as instruções sucessoras válidas e úteis. Diversas otimizações são utilizadas. A Figura C.14 mostra as três maneiras como o atraso de desvio condicional pode ser escalonado.

As limitações no escalonamento com desvio adiado surgem (1) das restrições nas instruções que estão escalonadas para os slots de atraso e (2) da nossa capacidade de prever em tempo de compilação se um desvio provavelmente será tomado ou não. Para melhorar a capacidade de o compilador preencher os slots de atraso de desvio condicional, a maioria dos processadores com desvios condicionais introduziu um desvio de *cancelamento* ou *anulação*. Em um desvio de cancelamento, a instrução inclui a direção em que o desvio foi previsto. Quando o desvio se comporta como previsto, a instrução no slot de atraso de desvio condicional é simplesmente executada como aconteceria normalmente com um desvio adiado. Quando o desvio estiver previsto incorretamente, a instrução no slot de atraso de desvio condicional será simplesmente transformada em um no-op.

Desempenho dos esquemas de desvio

Qual é o desempenho efetivo de cada um desses esquemas? O ganho de velocidade efetivo do pipeline com penalidades de desvio, considerando um CPI ideal de 1, é

$$\text{Ganho de velocidade do pipeline} = \frac{\text{Profundidade do pipeline}}{1 + \text{Frequência de desvio} \times \text{Penalidade de desvio}}$$

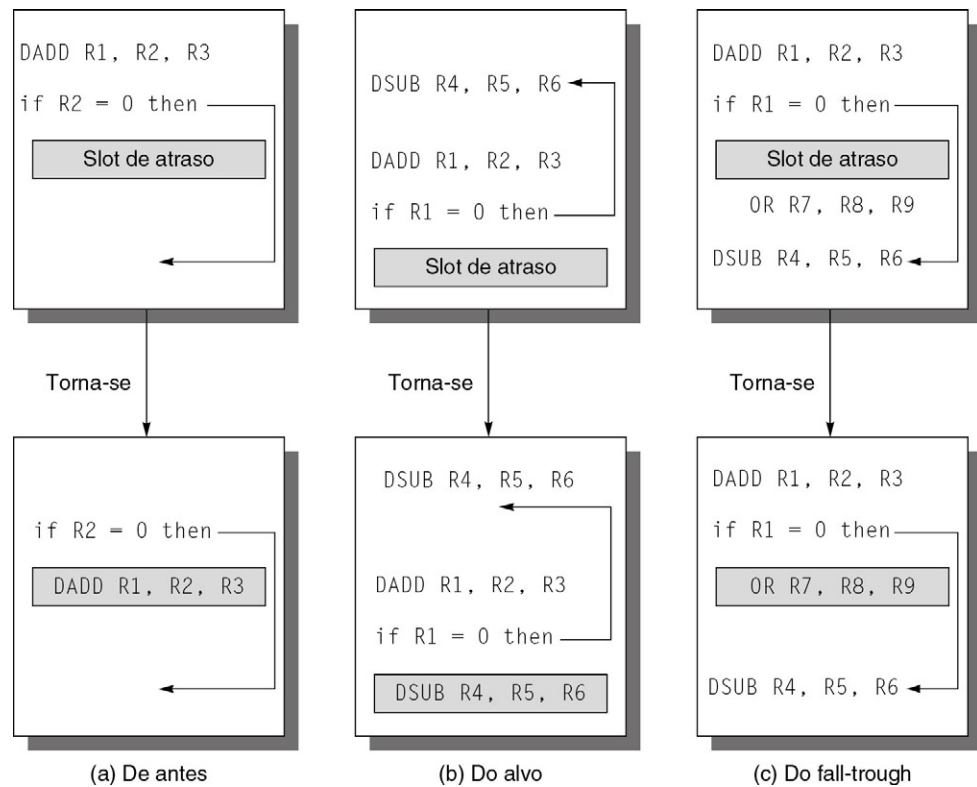



FIGURA C.14 **Escalonamento do slot de atraso de desvio condicional.**
A caixa superior em cada par mostra o código antes do escalonamento; a caixa inferior mostra o código escalonado. Em (a), o slot de atraso é escalonado com uma instrução independente de antes do desvio. Essa é a melhor opção. As estratégias (b) e (c) são usadas quando (a) não é possível. Nas sequências de código para (b) e (c), o uso de R1 na condição de desvio impede que a instrução DADD (cujo destino é R1) seja movida para após o desvio. Em (b), o slot de atraso de desvio condicional é escalonado a partir do destino do desvio; normalmente, a instrução de destino precisa ser copiada, pois pode ser alcançada por outro caminho. Estranhamente, (b) é preferido quando o desvio é tomado com alta probabilidade, como em um desvio de loop. Finalmente, o desvio pode ser escalonado a partir do fall-through não tomado, como em (c). Para tornar essa otimização válida para (b) ou (c), precisa ser OK executar a instrução movida quando o desvio seguir na direção não esperada. Com OK queremos dizer que o trabalho é desperdiçado, mas o programa ainda será executado corretamente. Isso acontece, por exemplo, em (c) se R7 for um registrador temporário não usado quando o desvio segue na direção inesperada.

Devido ao seguinte:

$$\text{Ciclos de stall do pipeline dos desvios} = \text{Frequência de desvio} \times \text{Penalidade de desvio}$$

obtemos

$$\text{Ganho de velocidade do pipeline} = \frac{\text{Profundidade do pipeline}}{1 + \text{Frequência de desvio} \times \text{Penalidade de desvio}}$$

A frequência de desvio e a penalidade de desvio podem ter um componente dos desvios incondicional e condicional. Porém, o último domina, pois é mais frequente.

Exemplo Para um pipeline mais profundo, como aquele em um MIPS R4000, leva-se pelo menos três estágios de pipeline antes que o endereço de destino de desvio seja conhecido, e um ciclo adicional antes que a condição de desvio seja avaliada, considerando que não haja stalls nos registradores na comparação condicional. Um atraso de três estágios leva às penalidades de desvio para os três esquemas de previsão mais simples, listados na Figura C.15.
Encontre a adição efetiva ao CPI que surge dos desvios para esse pipeline, considerando as seguintes frequências:

Desvio incondicional	4%
Desvio condicional, não tomado	6%
Desvio condicional, tomado	10%

Esquema de desvio	Penalidade incondicional	Penalidade não tomada	Penalidade tomada
Esvaziar pipeline	2	3	3
Previsto tomado	2	3	2
Previsto não tomado	2	0	3

FIGURA C.15 Penalidades de desvio para os três esquemas de previsão mais simples para um pipeline mais profundo.

	Acréscimos ao CPI vindos dos custos de desvio			
Esquema de desvio	Desvios incondicionais	Desvios condicionais não tomados	Desvios condicionais tomados	Todos os desvios
Frequência do evento	4%	6%	10%	20%
Pipeline com stall	0,08	0,18	0,30	0,56
Tomado previsto	0,08	0,18	0,20	0,46
Não tomado previsto	0,08	0,00	0,30	0,38

FIGURA C.16 Penalidades de CPI para os três esquemas de previsão de desvio e um pipeline mais profundo.

Resposta Achamos os CPIs multiplicando a frequência relativa dos desvios incondicional, condicional não tomado e condicional tomado pelas respectivas penalidades. Os resultados aparecem na Figura C.16.
As diferenças entre os esquemas são aumentadas substancialmente com esse atraso mais longo. Se o CPI básico fosse 1 e os desvios fossem a única fonte de stalls, o pipeline ideal seria 1,56 vez mais rápido do que um pipeline que usasse o esquema de pipeline com stall. O esquema previsto não tomado seria 1,13 vez melhor do que o esquema de pipeline com stall sob as mesmas suposições.

Reduzindo o custo dos desvios através da previsão

Conforme os pipelines ficam mais profundos e a penalidade potencial dos desvios aumenta, usar desvios adiados e esquemas similares se torna insuficiente. Em vez disso, precisamos nos voltar para modos mais agressivos de prever os desvios. Tais esquemas se encaixam em duas classes: esquemas estáticos de baixo custo, que dependem da informação disponível no momento da compilação, e estratégias que preveem desvios dinamicamente com base no comportamento do programa. Vamos discutir aqui as duas abordagens.

Previsão estática de desvio

O principal modo de melhorar a previsão de desvio no tempo de compilador é usar informações de perfil coletadas das execuções anteriores. A observação que faz isso valer a pena é de que, muitas vezes, o comportamento dos desvios é distribuído de modo bimodal. Ou seja, um desvio individual tem forte tendência a ser tomado ou não. A Figura C.17 mostra o sucesso da previsão de desvio usando essa estratégia. Os mesmos dados de entrada foram usados para as execuções e para coletar o perfil. Outros estudos mostraram que é importante mudar as entradas para que um perfil de uma execução diferente leve a somente pequena mudança na previsão baseada em perfil.

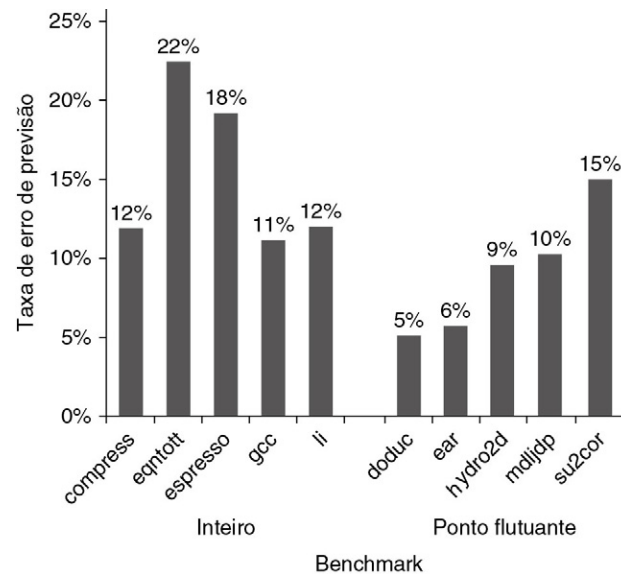



FIGURA C.17 A taxa de previsão incorreta no SPEC92 para um previsor baseado em perfil varia bastante, mas geralmente é melhor para os programas de ponto flutuante, que têm taxa média de previsão incorreta de 9% com desvio-padrão de 4%, do que para os programas de inteiros, que têm taxa média de previsão incorreta de 15% com desvio-padrão de 5%.
O desempenho real depende da precisão de previsão e da frequência de desvio, que variam entre 3-24%.

A efetividade de qualquer esquema de previsão de desvio depende da precisão do esquema e da frequência dos desvios condicionais, que variam na SPEC de 3-24%. O fato de que a taxa de previsão incorreta para os programas inteiros é maior e de que esses programas geralmente têm frequência de desvio maior é uma grande limitação para a previsão estática de desvio. Na próxima seção, consideraremos os previsores dinâmicos de desvio, que os processadores mais recentes têm empregado.

Previsão dinâmica de desvio e buffers de previsão de desvio

O esquema de previsão dinâmica de desvio é um *buffer de previsão de desvio* ou *tabela de histórico de desvios*. Um buffer de previsão de desvio é uma pequena memória indexada pela porção inferior do endereço da instrução de desvio. A memória contém um bit que diz se o desvio foi tomado recentemente ou não. Esse esquema é o tipo de buffer mais simples. Ele não possui tags e é útil somente para reduzir o adiamento do desvio quando este é maior do que o tempo para calcular os possíveis PCs-alvo.

Com um buffer assim, não sabemos de fato se a previsão está correta — ela pode ter sido colocada lá por outro desvio que tenha os mesmos bits de endereço de baixa ordem. Mas

isso não importa. A previsão é uma dica que se supõe estar correta, e a busca começa na direção prevista. Se a dica estiver errada, o bit de previsão será invertido e armazenado.

Esse buffer é efetivamente uma cache em que cada acesso é um acerto, e, como veremos, o desempenho do buffer depende da frequência em que a previsão é feita para o desvio em que estamos interessados e da precisão dessa previsão quando há correspondência. Antes de analisarmos o desempenho, é útil fazer uma pequena mas importante melhoria na precisão do esquema de previsão de desvio.

Esse esquema simples de previsão com 1 bit tem uma desvantagem no desempenho: mesmo que um desvio seja quase sempre tomado, provavelmente vamos fazer uma previsão errada duas vezes quando ele não for tomado, já que a previsão incorreta faz com que o bit de previsão seja mudado.

Para solucionar esse ponto fraco, muitas vezes são usados esquemas de previsão com 2 bits. Em um esquema com 2 bits, uma previsão precisa estar errada duas vezes antes de ser mudada. A Figura C.18 mostra um processador de estado finito para um esquema de previsão de 2 bits.

Um buffer de previsão de desvio pode ser implementado com uma pequena "cache" especial acessada com o endereço de instrução durante o estágio do pipe IF ou como um par de bits ligado a cada bloco na cache de instrução e buscado com a instrução. Se a instrução for decodificada como um desvio e se o desvio for previsto como sendo tomado, a busca começará a partir do alvo assim que o PC for conhecido. Caso contrário, a busca e a execução sequenciais continuam. Como a Figura C.18 mostra, se a previsão estiver errada, os bits de previsão serão mudados.

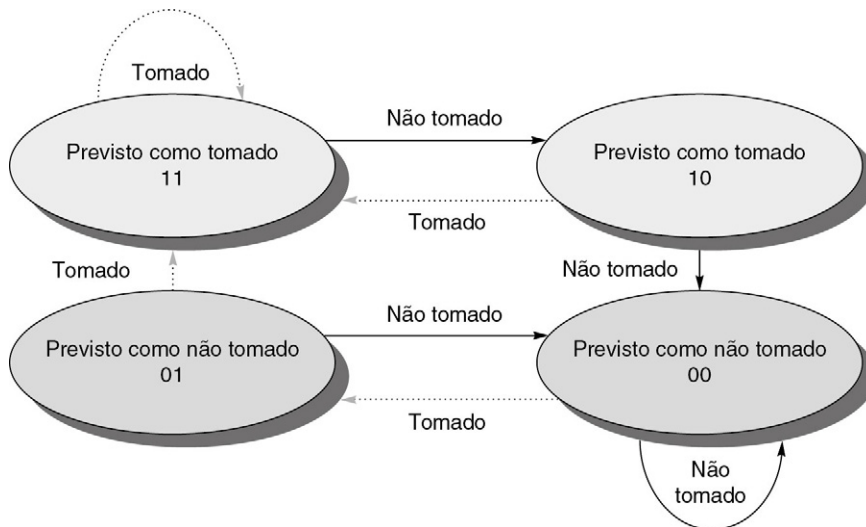



FIGURA C.18 Os estados em um esquema de previsão de 2 bits.

Ao usar 2 bits em vez de 1, um desvio que favoreça fortemente tomado ou não tomado — como fazem muitos desvios — será previsto incorretamente com menos frequência do que um previsor de 1 bit. Os 2 bits são usados para codificar os quatro estados do sistema. O esquema de 2 bits, na verdade, é uma especialização de um esquema mais geral que tem um contador de saturação de *n* bits para cada entrada no buffer de previsão. Com um contador de *n* bits, o contador pode assumir valores entre 0 e $2^n - 1$: quando o contador é maior ou igual à metade do seu valor máximo ($2^n - 1$), o desvio é previsto como tomado. Caso contrário, ele é previsto como não tomado. Estudos sobre os previsores de *n* bits mostraram que os previsores de 2 bits têm desempenho quase tão bom e, assim, a maioria dos sistemas despende em previsores de desvio de 2 bits em vez dos previsores de *n* bits mais gerais.

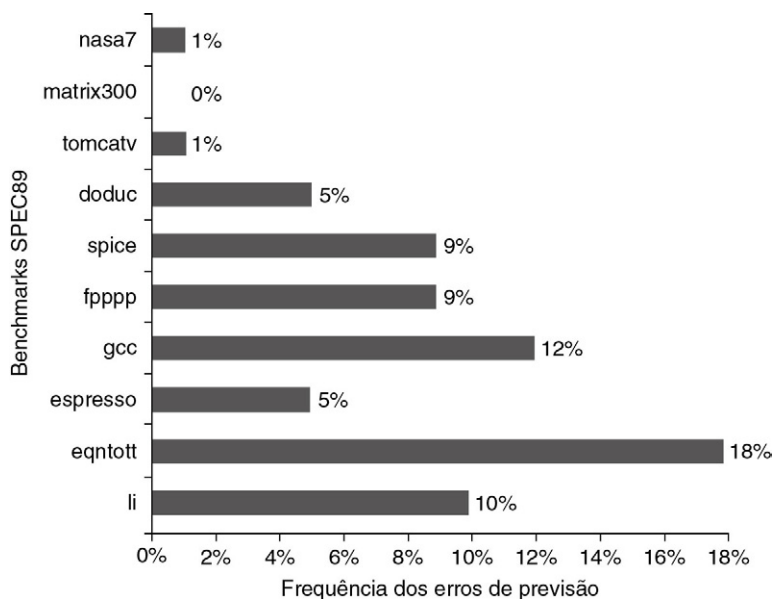



FIGURA C.19 Previsão de precisão de um buffer de 2 bits com 4.096 entradas para os benchmarks SPEC89. A taxa de erro de previsão para os benchmarks inteiros (gcc, espresso, egntott e li) é substancialmente maior (média de 11%) do que aquela para os programas de ponto flutuante (média de 4%). Omitir os kernels de ponto flutuante (nasa7, matrix300 e tomcaty) ainda gera uma precisão maior para os benchmarks de PF do que para os benchmarks inteiros. Esses dados, assim como o resto dos dados nesta seção, foram tirados de um estudo sobre previsão de desvio realizado usando a arquitetura Power da IBM e código otimizado para esse sistema. Veja Pan et al., (1992). Embora esses dados sejam para uma versão mais antiga do subconjunto dos benchmarks SPEC, os benchmarks mais novos são maiores e vão apresentar comportamento ligeiramente pior, especialmente para os benchmarks inteiros.

Que tipo de precisão pode-se esperar de um buffer de previsão de desvio usando 2 bits por entrada em aplicações reais? A Figura C.19 mostra que para os benchmarks SPEC89 um buffer de previsão de desvio com 4.096 entradas resulta em uma previsão de previsão variando de mais de 99% a 82%, ou uma *taxa de erro de previsão* de 1-18%. Um buffer de entrada de 4 K, como o usado para esses resultados, é considerado pequeno pelos padrões de 2005, e um buffer maior poderia produzir resultados um pouco melhores.

Conforme tentamos explorar mais ILP, a precisão da nossa previsão de desvio se torna fundamental. Como podemos ver na Figura C.19, a precisão dos previsores para programas inteiros, que costumam ter frequências de desvio maiores, é menor do que para os programas científicos, intensos em termos de loops. Podemos atacar esse problema de dois modos: aumentando o tamanho do buffer ou aumentando a precisão do esquema que usamos para cada previsão. Um buffer com 4 K entradas, entretanto, como mostra a Figura C.20, tem desempenho comparável a um buffer infinito, pelo menos para os benchmarks como aqueles no SPEC. Os dados da Figura C.20 deixam claro que a taxa de acerto do buffer não é o principal fator limitante. Como mencionamos, simplesmente aumentar o número de bits por previsto, sem mudar a estrutura do previsor, também tem pouco impacto. Em vez disso, precisamos examinar como podemos aumentar a precisão de cada previsor.

C.3 COMO O PIPELINING É IMPLEMENTADO?

Antes de prosseguirmos para o pipelining básico, precisamos rever uma implementação simples de uma versão do MIPS sem pipeline.

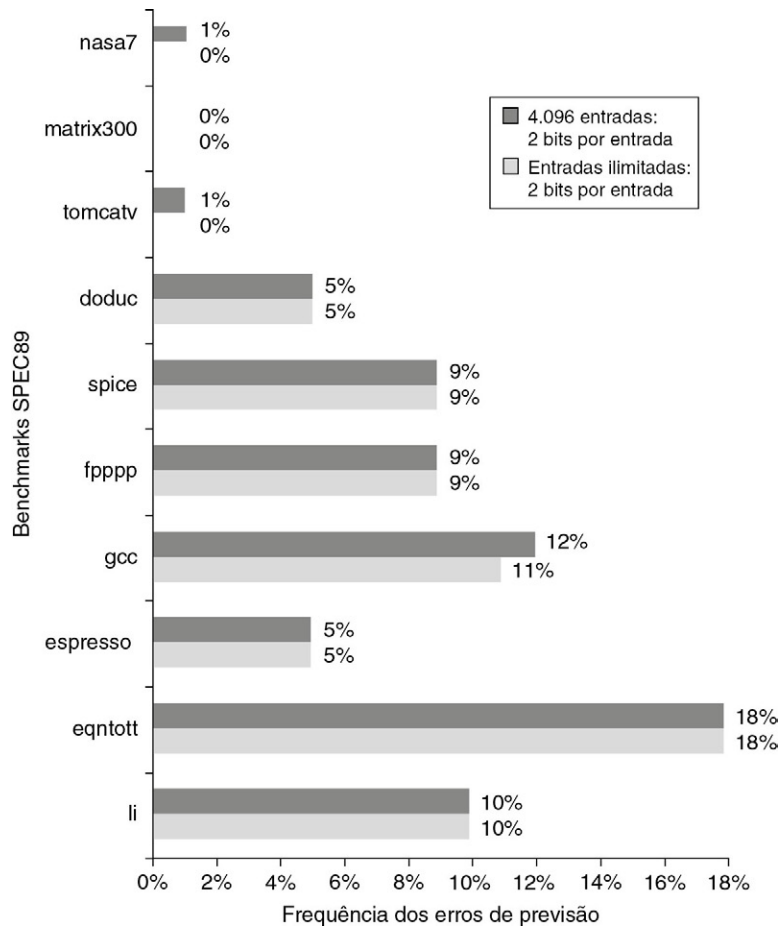



FIGURA C.20 **Previsão de precisão de um buffer de 2 bits com 4.096 entradas em comparação a um buffer infinito para os benchmarks SPEC89.**

Embora esses dados sejam para uma versão mais antiga de um subconjunto dos benchmarks SPEC, os resultados seriam comparáveis para versões mais novas, talvez com o mesmo número de entradas de 8 K necessárias para corresponder a um previsor infinito de 2 bits.

Uma implementação simples do MIPS

Nesta seção, seguimos o estilo da Seção C.1, mostrando primeiro uma implementação sem pipeline simples e depois uma com pipeline. Porém, desta vez, nosso exemplo é específico para a arquitetura MIPS.

Nesta subseção, enfocaremos um pipeline para um subconjunto de inteiros do MIPS, que consiste em "load-store de palavra", "branch equal zero" e operações da ALU com inteiros. Mais adiante neste apêndice, incorporaremos as operações básicas de ponto flutuante. Embora discutamos apenas um subconjunto do MIPS, os princípios básicos podem ser estendidos para lidar com todas as instruções. Inicialmente, usamos uma implementação menos agressiva de uma instrução de desvio. Mostramos como implementar a versão mais agressiva ao final desta seção.

Cada instrução do MIPS pode ser implementada, no máximo, em cinco ciclos de clock. Os cinco ciclos de clock são os seguintes:

1. Ciclo de busca de instrução (IF):

```
IR ← Mem[PC];
NPC ← PC + 4;
```

Operação: Enviar o PC e buscar a instrução da memória para o registrador de instruções (IR), incrementar o PC em 4 para endereçar a próxima instrução sequencial. O IR é usado para manter a instrução que será necessária nos ciclos de clock subsequentes; da mesma forma, o NPC do registrador é usado para manter o próximo PC sequencial.

2. *Decodificador de instrução/ciclo de busca de registrador* (ID):

```
A ← Regs[rs];
B ← Regs[rt];
Imm ← campo imediato estendido por sinal do IR;
```

Operação: Decodificar a instrução e acessar o banco de registradores para ler os registradores (rs e rt são os especificadores de registrador). As saídas dos registradores de uso geral são lidas para dois registradores temporários (A e B) para uso em outros ciclos de clock. Os 16 bits inferiores do IR também são estendidos com o valor do sinal e armazenados no registrador temporário Imm, para uso no próximo ciclo de clock.

A decodificação é feita em paralelo com a leitura de registradores, o que é possível porque esses campos estão em um local fixo no formato de instrução do MIPS. Como a parte imediata de uma instrução está localizada em um local idêntico em cada formato MIPS, o imediato estendido com o valor do sinal também é calculado durante esse ciclo, caso seja necessário no próximo ciclo.

3. *Execução/ciclo de endereço efetivo* (EX):

A ALU opera sobre os operandos preparados no ciclo anterior, realizando uma das quatro funções, dependendo do tipo de instrução MIPS.

- Referência de memória:

```
ALUOutput ← A + Imm;
```

Operação: A ALU soma os operandos para formar o endereço efetivo e coloca o resultado no registrador ALUOutput.

- Instrução da ALU registrador-registrador:

```
ALUOutput←A func B;
```

Operação: A ALU realiza a operação especificada pelo código de função no valor do registrador A e no valor do registrador B. O resultado é colocado no registrador temporário ALUOutput.

- Instrução da ALU registrador-imediato:

```
ALUOutput←A op Imm;
```

Operação: A ALU realiza a operação especificada pelo opcode sobre o valor no registrador A e sobre o valor no registrador Imm. O resultado é colocado no registrador temporário ALUOutput.

- Desvio:

```
ALUOutput ← NPC + (Imm << 2);
Cond ← (A == 0)
```

Operação: A ALU soma o NPC ao valor imediato estendido com o valor do sinal em Imm, que é deslocado à esquerda por 2 bits para criar um offset de palavra, a fim de calcular o endereço do destino do desvio. O registrador A, que foi lido no ciclo anterior, é verificado para se determinar se o desvio foi tomado. Como estamos considerando apenas uma forma de desvio (BEQZ), a comparação é contra 0. Observe que BEQZ é, na realidade, uma pseudoinstrução que se traduz em um BEQ com R0 como operando. Por simplicidade, essa é a única forma de desvio que consideramos.

A arquitetura load-store do MIPS significa que o endereço efetivo e os ciclos de execução podem ser combinados em um único ciclo de clock, pois nenhuma instrução precisa calcular simultaneamente um endereço de dados, calcular um endereço de destino de instrução e realizar uma operação sobre os dados. As outras instruções com inteiros não incluídas anteriormente são saltos de várias formas, semelhantes a desvios condicionais.

4. *Acesso à memória/ciclo de término de desvio* (MEM):

O PC é atualizado para todas as instruções: PC ← NPC;

- Referência à memória:

```
LMD ← Mem[ALUOutput] ou
Mem[ALUOutput] ← B;
```

Operação: Acessar a memória, se necessário. Se a instrução for um load, os dados são lidos da memória e são colocados no registrador LMD (Load Memory Data); se for um store, os dados do registrador B são escritos na memória. De qualquer forma, o endereço usado é aquele calculado durante o ciclo anterior e armazenado no registrador ALUOutput.

- Desvio:

```
if (cond) PC ← ALUOutput
```

Operação: Se a instrução desviar, o PC é substituído pelo endereço de destino do desvio no registrador ALUOutput.

5. *Ciclo de write-back* (WB):

- Instrução da ALU registrador-registrador:

```
Regs[rd] ← ALUOutput;
```

- Instrução da ALU registrador-imediato:

```
Regs[rt] ← ALUOutput;
```

- Instrução load:

```
Regs[rt] ← LMD;
```

Operação: Escrever o resultado no banco de registradores, venha ele do sistema de memória (que está no registrador LMD) ou da ALU (que está em ALUOutput); o campo de destino do registrador também está em uma de duas posições (rd ou rt), dependendo do opcode efetivo.

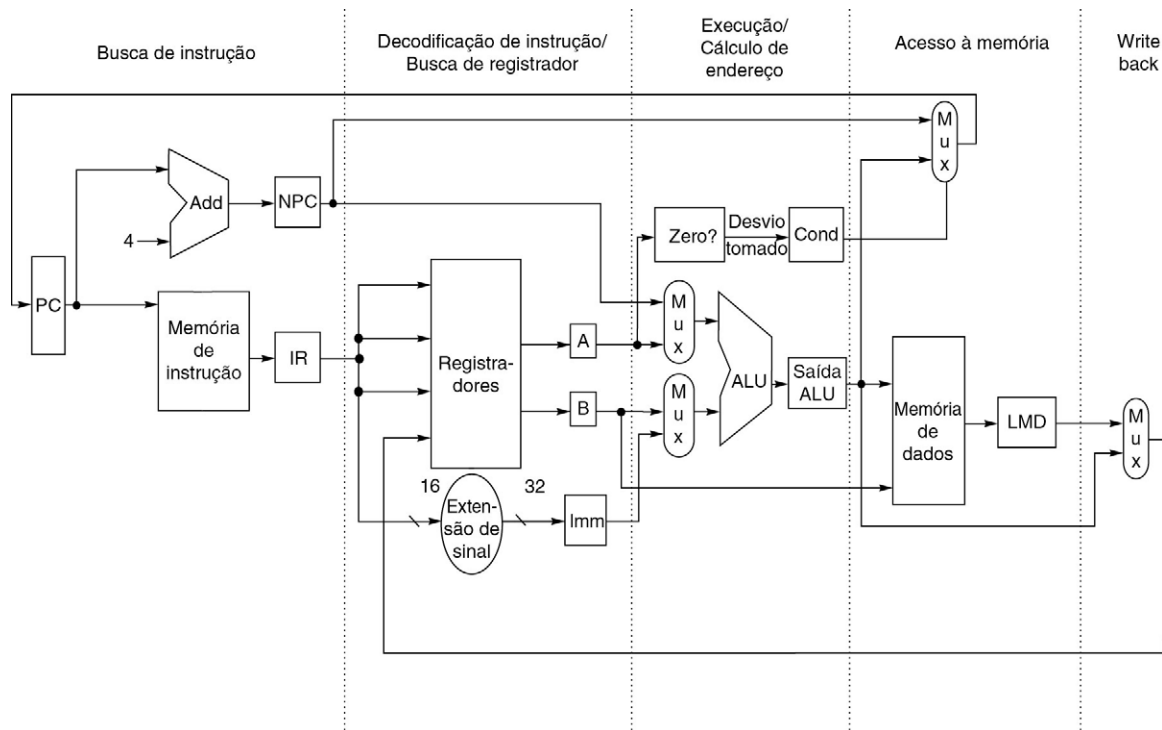



FIGURA C.21 **A implementação do datapath do MIPS permite que cada instrução seja executada em quatro ou cinco ciclos de clock.** Embora o PC seja mostrado na parte do datapath usado na busca de instrução e os registradores apareçam na parte do datapath usado na decodificação de instrução/busca no registrador, ambas as unidades funcionais são lidas e escritas por uma instrução. Embora mostremos essas unidades funcionais no ciclo correspondente onde elas são lidas, o PC é escrito durante o ciclo de clock de acesso à memória, e os registradores são escritos durante o ciclo de clock de write-back. Nos dois casos, as escritas em outros estágios de pipe posteriores são indicadas pela saída do multiplexador (no acesso à memória ou write-back), que transporta um valor de volta ao PC ou aos registradores. Esses sinais fluindo ao contrário introduzem grande parte da complexidade do pipelining, pois indicam a possibilidade de hazards.

A Figura C.21 mostra como uma instrução flui pelo datapath. Ao final de cada ciclo de clock, cada valor calculado durante esse ciclo de clock e requisitado em um ciclo de clock posterior (seja para essa instrução seja para a seguinte) é gravado em um dispositivo de armazenamento, que pode ser a memória, um registrador de uso geral, o PC ou um registrador temporário (ou seja, LMD, Imm, A, B, IR, NPC, ALUOutput ou Cond). Os registradores temporários mantêm valores entre os ciclos de clock para uma instrução, enquanto os outros elementos de armazenamento são partes visíveis do estado e mantêm valores entre instruções sucessivas.

Embora hoje todos os processadores tenham pipeline, essa implementação multiciclos é uma aproximação razoável de como a maioria dos processadores teria sido implementada em épocas anteriores. Uma máquina de estado finito simples poderia ser usada para implementar o controle seguindo a estrutura de cinco ciclos mostrada. Para um processador muito mais complexo, o controle com microcódigo poderia ser usado. De qualquer forma, uma sequência de instruções como a anterior determinaria a estrutura do controle.

Existem algumas redundâncias de hardware que poderiam ser eliminadas nessa implementação multiciclos. Por exemplo, existem duas ALUs: uma para incrementar o PC e outra usada para o endereço efetivo e cálculo da ALU. Como elas não são necessárias no mesmo ciclo de clock, poderíamos mesclá-las acrescentando multiplexadores adicionais e compartilhando a mesma ALU. De modo semelhante, instruções e dados poderiam ser

armazenados na mesma memória, pois os acessos a dados e instruções acontecem em ciclos de clock diferentes.

Em vez de otimizar essa implementação simples, deixaremos o projeto como está na Figura C.21, pois isso nos dá uma base melhor para a implementação em pipeline.

Como alternativa ao projeto de multiciclos discutido nesta seção, também poderíamos ter implementado a CPU de modo que cada instrução ocupe um ciclo de clock. Nesse caso, os registradores temporários seriam excluídos, pois não haveria qualquer comunicação entre os ciclos de clock dentro de uma instrução. Cada instrução seria executada em um ciclo de clock, escrevendo o resultado na memória de dados, registradores ou PC ao final do ciclo de clock. O CPI seria único para tal processador. O ciclo de clock, porém, seria aproximadamente igual a cinco vezes o ciclo de clock do processador multiciclos, pois cada instrução teria que atravessar todas as unidades funcionais. Os projetistas nunca usariam essa implementação de único ciclo por dois motivos: 1) uma implementação de único ciclo seria muito ineficaz para a maioria das CPUs que possuem uma variação razoável entre a quantidade de trabalho — portanto, no tempo de ciclo de clock — necessária para diferentes instruções; 2) uma implementação de único ciclo exige a duplicação de unidades funcionais que poderiam ser compartilhadas em uma implementação multiciclos. Apesar disso, esse datapath de único ciclo nos permite ilustrar como o pipelining pode melhorar o tempo do ciclo de clock de um processador, ao contrário do CPI.

Um pipeline básico para o MIPS

Como antes, podemos implementar um pipeline no datapath da Figura C.21 quase sem mudanças, iniciando uma nova instrução em cada ciclo de clock. Como cada estágio de pipe está ativo em cada ciclo de clock, todas as operações em um estágio de pipe precisam concluir em um ciclo de clock, e qualquer combinação de operações precisa ser capaz de ocorrer ao mesmo tempo. Além do mais, o pipelining do datapath requer que os valores passados de um estágio de pipe para o seguinte devam ser colocados em registradores. A Figura C.22 mostra o pipeline MIPS com os registradores apropriados, chamados *registradores de pipeline* ou *latches de pipeline*, entre cada estágio do pipeline. Os registradores são rotulados com os nomes dos estágios a que se conectam. A Figura C.22 é desenhada de modo que as conexões através dos registradores de pipeline de um estágio para outro sejam claras.

Todos os registradores necessários para manter valores temporariamente entre os ciclos de clock dentro de uma instrução estão incluídos nesses registradores de pipeline. Os campos do registrador de instruções (IR) — o qual faz parte do registrador IF/ID — são rotulados quando usados para fornecer nomes de registradores. Os registradores de pipeline transportam dados e controle de um estágio do pipeline para o seguinte. Qualquer valor necessário em um estágio posterior do pipeline precisa ser colocado em tal registrador e copiado de um registrador do pipeline para o seguinte, até que não seja mais necessário. Se tentássemos simplesmente usar os registradores temporários que tínhamos em nosso datapath sem pipeline anterior, os valores poderiam ser modificados antes que todos os usos fossem concluídos. Por exemplo, o campo de um operando de registrador usado para uma escrita em um load ou operação da ALU é fornecido a partir do registrador de pipeline MEM/WB, em vez do registrador IF/ID. Isso porque queremos que um load ou uma operação da ALU escreva no registrador designado por essa operação, e não no campo de registrador da instrução atualmente fazendo a transição de IF para ID! Esse campo de registrador de destino é simplesmente copiado de um registrador de pipeline para o seguinte, até que seja necessário durante o estágio WB.

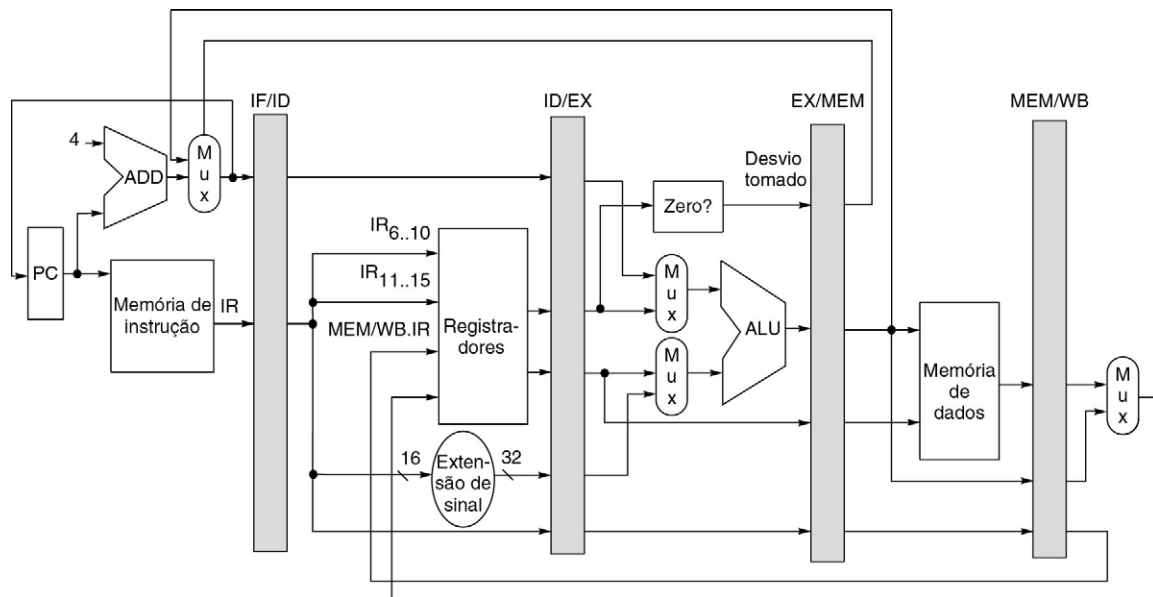



FIGURA C.22 O datapath é implemenatdo com pipeline pela inclusão de um conjunto de registradores, um entre cada par de estágios de pipe. Os registradores servem para conduzir valores e informações de controle de um estágio para o próximo. Também podemos pensar no PC como um registrador de pipeline, que fica antes do estágio IF do pipeline, levando a um registrador de pipeline para cada estágio de pipe. Lembre-se de que o PC é um registrador acionado pela borda, escrito ao final do ciclo de clock; daí não existir condição de race (corrida) na escrita do PC. O multiplexador de seleção para o PC foi movido de modo que o PC seja escrito exatamente em um estágio (IF). Se não o movêssemos, haveria um conflito quando ocorresse um desvio, pois duas instruções tentariam escrever valores diferentes no PC. A maior parte das vias de dados flui da esquerda para a direita, que é de um ponto anterior no tempo para outro posterior. As vias fluindo da direita para a esquerda (que transportam a informação de write-back do registrador e a informação do PC em um desvio) introduzem complicações ao nosso pipeline.

Qualquer instrução está ativa em exatamente um estágio do pipeline de cada vez; portanto, quaisquer ações tomadas em favor de uma instrução ocorrem entre um par de registradores de pipeline. Assim, também podemos ver as atividades do pipeline examinando o que precisa acontecer em qualquer estágio do pipeline, dependendo do tipo de instrução. A Figura C.23 mostra essa visão. Os campos dos registradores de pipeline são nomeados de modo a mostrar o fluxo de dados de um estágio para o seguinte. Observe que as ações nos dois primeiros estágios são independentes do tipo de instrução atual; elas precisam ser independentes, porque a instrução não é decodificada antes do final do estágio ID. A atividade IF depende de a instrução em EX/MEM ser um desvio tomado. Se for, então o endereço de destino de desvio para a instrução de desvio em EX/MEM é escrito no PC ao final do IF; caso contrário, o PC incrementado será escrito de volta. (Como já dissemos, esse efeito de desvios condicionais ocasiona complicações no pipeline, das quais trataremos nas próximas seções.) A codificação de posição fixa dos operandos dos registradores é crítica para permitir que os registradores sejam lidos durante o ID.

Para controlar esse pipeline simples, só precisamos determinar como configurar o controle para os quatro multiplexadores no datapath da Figura C.22. Os dois multiplexadores no estágio da ALU são definidos de acordo com o tipo de instrução, que é ditado pelo campo IR do registrador ID/EX. O multiplexador da entrada superior da ALU é definido dependendo de a instrução ser um desvio ou não, e o multiplexador inferior é definido dependendo de a instrução ser uma operação da ALU registrador-registrador ou qualquer outro tipo de operação. O multiplexador no estágio IF escolhe se vai usar o valor do PC incrementado ou o valor do EX/MEM.ALUOutput (o destino do desvio) para escrever no PC. Esse multiplexador é controlado pelo campo EX/MEM.cond. O quarto multiplexador

Estágio	Qualquer instrução		
IF	IF/ID.IR ← Mem[PC]; IF/ID.NPC,PC ← (if ((EX/MEM.opcode == branch) & EX/MEM.cond){EX/MEM. ALUOutput} else {PC+4});		
ID	ID/EX.A ← Regs[IF/ID.IR[rs]]; ID/EX.B ← Regs[IF/ID.IR[rt]]; ID/EX.NPC ← IF/ID.NPC; ID/EX.IR←IF/ID.IR; ID/EX.Imm ← sign-extend(IF/ID.IR[immediate field]);		
	Instrução da ALU	**Instrução load ou store**	**Instrução de desvio**
EX	EX/MEM.IR ← ID/EX.IR; EX/MEM.ALUOutput ← ID/EX.A func ID/EX.B; ou EX/MEM.ALUOutput ← ID/EX.A op ID/EX.Imm;	EX/MEM.IR to ID/EX.IR EX/MEM.ALUOutput ← ID/EX.A + ID/EX.Imm; EX/MEM.B←ID/EX.B;	EX/MEM.ALUOutput ← ID/EX.NPC+ (ID/EX.Imm << 2); EX/MEM.cond ← (ID/EX.A == 0);
MEM	MEM/WB.IR ← EX/MEM.IR; MEM/WB.ALUOutput ← EX/MEM.ALUOutput;	MEM/WB.IR ← EX/MEM.IR; MEM/WB.LMD ← Mem[EX/MEM.ALUOutput]; ou Mem[EX/MEM.ALUOutput] ← EX/MEM.B;	
WB	Regs[MEM/WB.IR[rd]] ← MEM/WB.ALUOutput; ou Regs[MEM/WB.IR[rt]] ← MEM/WB.ALUOutput;	For load only:nly: Regs[MEM/WB.IR[rt]] ← MEM/WB.LMD;	

FIGURA C.23 Eventos em cada estágio de pipe do pipeline MIPS.

Vamos rever as ações nos estágios que são específicos à organização do pipeline. Em IF, além de buscar a instrução e calcular o novo PC, armazenamos o PC incrementado tanto no PC quanto em um registrador de pipeline (NPC) para uso posterior no cálculo do endereço de destino do desvio. Essa estrutura é a mesma que a organização na Figura C.22, onde o PC é atualizado no IF a partir de uma ou duas origens. Em ID, buscamos os registradores, estendemos o sinal dos 16 bits inferiores do IR (o campo imediato) e passamos o IR e o NPC. Durante EX, realizamos uma operação da ALU ou um cálculo de endereço; passamos o IR e o registrador B (se a instrução for um store). Também definimos o valor de cond como 1 se a instrução for um desvio tomado. Durante a fase MEM, alternamos a memória, escrevemos o PC, se for necessário, e passamos os valores necessários no estágio de pipe final. Finalmente, durante WB, atualizamos o campo do registrador a partir da saída da ALU ou do valor carregado. Para simplificar, sempre passamos o IR inteiro de um estágio para o seguinte, embora, à medida que uma instrução prossegue pelo pipeline, menos e menos do IR é necessário.

é controlado dependendo de a instrução no estágio WB ser um load ou uma operação da ALU. Além desses quatro multiplexadores, existe um multiplexador adicional necessário, que não aparece desenhado na Figura C.22, mas cuja existência é clara, olhando para o estágio WB de uma operação da ALU. O campo de registrador de destino é um de dois lugares diferentes, de acordo com o tipo de instrução (ALU registrador-registrador contra ALU imediato ou load). Assim, precisaremos de um multiplexador para escolher a parte correta do IR no registrador MEM/WB a fim de especificar o campo de destino do registrador, considerando que a instrução escreve em um registrador.

Implementando o controle para o pipeline MIPS

O processo de permitir que uma instrução se mova do estágio de decodificação de instrução (ID) para o estágio de execução (EX) desse pipeline normalmente é chamado *despacho de instrução*; uma instrução que fez essa etapa é considerada como *despachada*. Para o pipeline de inteiros do MIPS, todos os hazards de dados podem ser verificados durante a fase ID do pipeline. Se houver um hazard de dados, a instrução é adiada antes de ser despachada.

De modo semelhante, podemos determinar que adiantamento será necessário durante ID e definir os controles apropriados. A detecção de interbloqueios antecipadamente no pipeline reduz a complexidade do hardware, pois o hardware nunca precisa suspender uma instrução que atualizou o estado do processador, a menos que o processador inteiro fique em stall. Como alternativa, podemos detectar o hazard ou o adiantamento no início de um ciclo de clock que usa um operando (EX e MEM para esse pipeline). Para mostrar as diferenças nessas duas técnicas, mostraremos o interbloqueio para um hazard RAW como uma origem vindo de uma instrução de load (chamado *interbloqueio de laod — load interlock*) pode ser implementado por uma verificação no ID, enquanto a implementação das vias de adiantamento para as entradas da ALU podem ser feitas durante EX. A Figura C.24 lista a variedade de circunstâncias com que precisamos lidar.

Situação	Sequência de código de exemplo	Ação

Sem dependência	`LD R1,45(R2)`
	`DADD R5,R6,R7`
	`DSUB R8,R6,R7`
	`OR R9,R6,R7`
		Nenhum hazard possível, pois não existe dependência em R1 nas três instruções imediatamente seguintes.

Dependência exigindo stall	`LD R1,45(R2)`
	`DADD R5,R1,R7`
	`DSUB R8,R6,R7`
	`OR R9,R6,R7`
		Comparadores detectam o uso de R1 no `DADD` e adiam o `DADD` (e `DSUB` e `OR`) antes que o `DADD` inicie EX.

Dependência contornada pelo adiantamento	`LD R1,45(R2)`
	`DADD R5,R6,R7`
	`DSUB R8,R1,R7`
	`OR R9,R6,R7`
		Comparadores detectam o uso de R1 em `DSUB` e encaminham o resultado do load para a ALU em tempo para `DSUB` iniciar EX.

Dependência com acessos em ordem	`LD R1,45(R2)`
	`DADD R5,R6,R7`
	`DSUB R8,R6,R7`
	`OR R9,R1,R7`
		Nenhuma ação exigida, pois a leitura de R1 por `OR` ocorre na segunda metade da fase ID, enquanto a escrita dos dados carregados ocorreu na primeira metade.

FIGURA C.24 Situações que o hardware de detecção de hazard do pipeline pode ver comparando o destino e as origens de instruções adjacentes.

Essa tabela indica que a única comparação necessária é entre o destino e as origens nas duas instruções após a instrução que escreveu no destino. No caso de um stall, as dependências do pipeline se parecerão com o terceiro caso quando a execução continuar. Naturalmente, os hazards que envolvem R0 podem ser ignorados, pois o registrador sempre contém 0, e o teste anterior poderia ser estendido para isso.

Vamos começar implementando o interbloqueio de load. Se houver um hazard RAW com a instrução de origem sendo um load, a instrução de load estará no estágio EX quando uma instrução que precisa dos dados de load estiver no estágio ID. Assim, podemos descrever todas as situações de hazard possíveis com uma pequena tabela, que pode ser traduzida diretamente para uma implementação. A Figura C.25 mostra uma tabela que detecta todos os interbloqueios de load quando a instrução usando o resultado do load estiver no estágio ID.

Quando um hazard tiver sido detectado, a unidade de controle precisará inserir o stall no pipeline e impedir que as instruções nos estágios IF e ID avancem. Como já dissemos, toda informação de controle é transportada nos registradores de pipeline (transportar a instrução é suficiente, pois todo o controle é derivado disso). Assim, quando detectarmos um hazard, só precisamos mudar a parte de controle do registrador de pipeline ID/EX para valores 0, que é uma no-op (uma instrução que não faz nada, como `DADD R0, R0, R0`).

Campo opcode de ID/EX (ID/EX.$IR_{0..5}$)	Campo opcode de IF/ID (IF/ID.$IR_{0..5}$)	Campos de operando combinando
Load	ALU registrador-registrador	`ID/EX.IR[rt] == IF/ID.IR[rs]`
Load	ALU registrador-registrador	`ID/EX.IR[rt] == IF/ID.IR[rt]`
Load	Load, store, ALU imediato ou desvio	`ID/EX.IR[rt] == IF/ID.IR[rs]`

FIGURA C.25 **A lógica para detectar a necessidade de interbloqueios de load durante o estágio ID de uma instrução exige três comparações.**
As linhas 1 e 2 da tabela testam se o registrador de destino do load é um dos registradores-fonte para uma operação registrador-registrador em ID. A linha 3 da tabela determina se o registrador de destino do load é uma origem para endereço efetivo de load ou store, um imediato da ALU ou um teste de desvio. Lembre-se de que o registrador IF/ID mantém o estado da instrução em ID, que potencialmente utiliza o resultado do load, enquanto ID/EX mantém o estado da instrução em EX, que é a instrução de load.

Além disso, simplesmente recirculamos o conteúdo dos registradores IF/ID para manter a instrução adiada. Em um pipeline com hazards mais complexos, as mesmas ideias se aplicariam. Podemos detectar o hazard comparando algum conjunto de registradores de pipeline e deslocando no-ops para impedir a execução errônea.

A implementação da lógica de adiantamento é semelhante, embora existam mais casos a considerar. A principal observação necessária para implementar a lógica de endereço é de que os registradores de pipeline contenham os dados a serem encaminhados e também os campos do registrador-fonte e destino. Todo o adiantamento logicamente acontece da ALU ou da saída da memória de dados para a entrada da ALU, a entrada da memória de dados ou a unidade de detecção de zero. Assim, podemos implementar o adiantamento com uma comparação dos registradores de destino do IR contidos nos estágios EX/MEM e MEM/WB contra os registradores-fonte do IR contidos nos registradores ID/EX e EX/MEM. A Figura C.26 mostra as comparações e possíveis operações de adiantamento, onde o destino do resultado encaminhado é uma entrada da ALU para a instrução atualmente em EX.

Além dos comparadores e da lógica combinatória que precisamos determinar quando uma via de adiantamento precisa ser habilitada, também precisamos ampliar os multiplexadores nas entradas da ALU e acrescentar as conexões dos registradores de pipeline que são usados para encaminhar os resultados. A Figura C.27 mostra os segmentos relevantes do datapath em pipeline com os multiplexadores e as conexões adicionais.

Para o MIPS, o hardware de detecção de hazard e adiantamento é razoavelmente simples; veremos que as coisas se tornam um pouco mais complicadas quando estendemos esse pipeline para lidar com ponto flutuante. Antes de fazermos isso, precisamos tratar dos desvios condicionais.

Tratando dos desvios condicionais no pipeline

No MIPS, os desvios condicionais (`BEQ` e `BNE`) exigem o teste de um registrador pela igualdade com outro registrador, que pode ser R0. Se considerarmos apenas os casos de `BEQZ` e `BNEZ`, que exigem um teste de zero, é possível completar essa decisão ao final do ciclo ID movendo o teste de zero para esse ciclo. Para tirar proveito de uma decisão antecipada sobre se o desvio é tomado, os dois PCs (tomado e não tomado) precisam ser calculados antecipadamente. Calcular o endereço de destino de desvio durante ID requer

Registrador de pipeline contendo instrução de origem	Opcode da instrução de origem	Registrador de pipeline contendo instrução de destino	Opcode da instrução de destino	Destino do resultado adiantado	Comparação (se igual então encaminha)
EX/MEM	ALU registrador-registrador	ID/EX	ALU registrador-registrador, ALU imediato, load, store, branch	Entrada superior da ALU	`EX/MEM.IR[rd] == ID/EX.IR[rs]`
EX/MEM	ALU registrador-registrador	ID/EX	ALU registrador-registrador	Entrada inferior da ALU	`EX/MEM.IR[rd] == ID/EX.IR[rt]`
MEM/WB	ALU registrador-registrador	ID/EX	ALU registrador-registrador, ALU imediato, load, store, branch	Entrada superior da ALU	`MEM/WB.IR[rd] == ID/EX.IR[rs]`
MEM/WB	ALU registrador-registrador	ID/EX	ALU registrador-registrador	Entrada inferior da ALU	`MEM/WB.IR[rd] == ID/EX.IR[rt]`
EX/MEM	ALU imediato	ID/EX	ALU registrador-registrador, ALU imediato, load, store, branch	Entrada superior da ALU	`EX/MEM.IR[rt] == ID/EX.IR[rs]`
EX/MEM	ALU imediato	ID/EX	ALU registrador-registrador	Entrada inferior da ALU	`EX/MEM.IR[rt] == ID/EX.IR[rt]`
MEM/WB	ALU imediato	ID/EX	ALU registrador-registrador, ALU imediato, load, store, branch	Entrada superior da ALU	`MEM/WB.IR[rt] == ID/EX.IR[rs]`
MEM/WB	ALU imediato	ID/EX	ALU registrador-registrador	Entrada inferior da ALU	`MEM/WB.IR[rt] == ID/EX.IR[rt]`
MEM/WB	Load	ID/EX	ALU registrador-registrador, ALU imediato, load, store, branch	Entrada superior da ALU	`MEM/WB.IR[rt] == ID/EX.IR[rs]`
MEM/WB	Load	ID/EX	ALU registrador-registrador	Entrada inferior da ALU	`MEM/WB.IR[rt] == ID/EX.IR[rt]`

FIGURA C.26 O adiantamento de dados às duas entradas da ALU (para as instruções em EX) pode ocorrer a partir do resultado da ALU (em EX/MEM ou em MEM/WB) ou do resultado do load em MEM/WB.

Existem 10 comparações separadas necessárias para saber se uma operação de adiantamento deve ocorrer. As entradas da ALU superior e inferior referem-se às entradas correspondentes ao primeiro e segundo operandos-fonte da ALU, respectivamente, e aparecem explicitamente nas Figuras C.21 e C.27. Lembre-se de que o latch do pipeline para a instrução de destino em EX é ID/EX, enquanto os valores-fonte vêm da parte ALUOutput de EX/MEM ou de MEM/WB, ou da parte LMD de MEM/WB. Existe uma complicação não resolvida por essa lógica: o tratamento de múltiplas instruções que escrevem no mesmo registrador. Por exemplo, durante a sequência de código `DADD R1, R2, R3; DADDI R1, R1, #2; DSUB R4, R3, R1`, a lógica precisa garantir que a instrução `DSUB` usa o resultado da instrução DADDI em vez do resultado da instrução `DADD`. A lógica mostrada pode ser estendida para lidar com esse caso simplesmente testando se o adiantamento de MEM/WB está habilitado apenas quando o adiantamento de EX/MEM não está habilitado para a mesma entrada. Como o resultado de `DADDI` estará em EX/MEM, ele será encaminhado, em vez do resultado de `DADD` em MEM/WB.

um somador adicional, pois a ALU principal, que tem sido usada para essa função até aqui, não pode ser usada antes de EX. A Figura C.28 mostra o datapath do pipeline revisado. Com o somador separado e uma decisão de desvio feita durante ID, existe apenas um stall de um ciclo de clock nos desvios. Embora isso reduza o atraso de desvio condicional para um ciclo, significa que uma instrução da ALU seguida por um desvio no resultado da instrução incorrerá em um stall de hazard de dados. A Figura C.29 mostra a parte de desvio da tabela de pipeline revisada da Figura C.23.

Em alguns processadores, os hazards de desvios condicionais são ainda mais dispendiosos nos ciclos de clock do que em nosso exemplo, pois o tempo para avaliar a condição de desvio e calcular o destino pode ser ainda maior. Por exemplo, um processador com

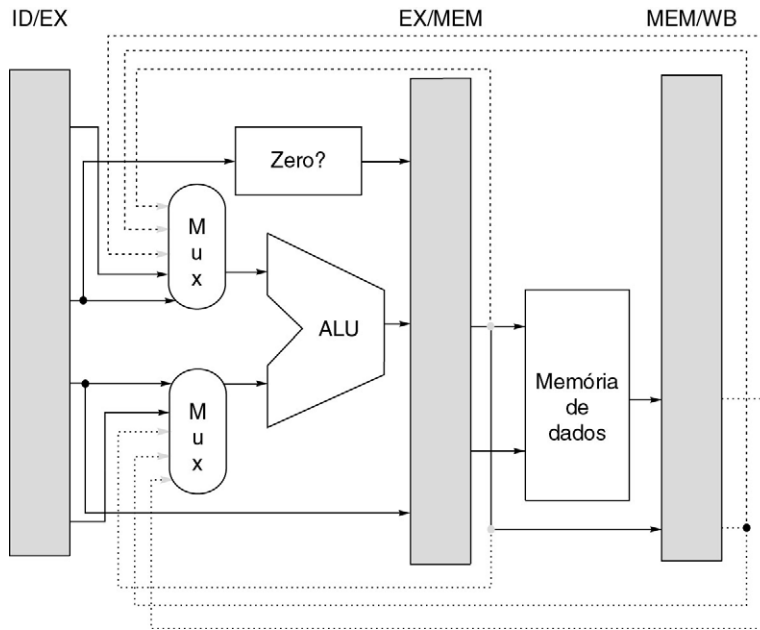



FIGURA C.27 O adiantamento de resultados para a ALU exige o acréscimo de três entradas extras em cada multiplexador da ALU e o acréscimo de três vias para as novas entradas.

As vias correspondem a um bypass de (1) saída da ALU no final do EX, (2) saída da ALU no final do estágio MEM e (3) saída da memória no final do estágio MEM.

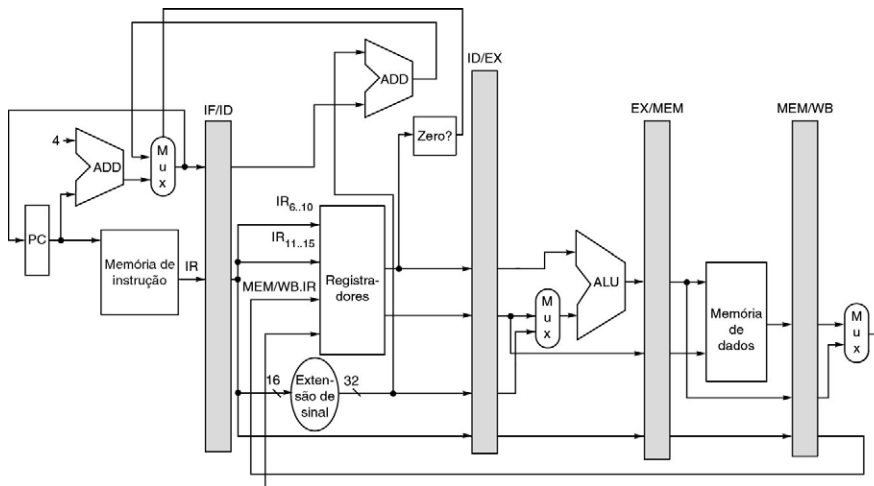



FIGURA C.28 O stall dos hazards de desvio condicionais podem ser reduzidos movendo-se o teste de zero e o cálculo do destino de desvio para a fase ID do pipeline.

Observe que fizemos duas mudanças importantes, cada uma removendo um ciclo do stall de três ciclos para os desvios condicionais. A primeira mudança é mover o cálculo do endereço de destino do desvio e a decisão da condição de desvio para o ciclo ID. A segunda mudança é escrever o PC da instrução na fase IF, usando o endereço de destino do desvio, calculado durante ID, ou o PC incrementado, calculado durante IF. Em comparação, a Figura C.22 obteve o endereço de destino do desvio do registrador EX/MEM e escreveu o resultado durante o ciclo de clock MEM. Conforme mencionamos na Figura C.22, o PC pode ser considerado como um registrador de pipeline (p. ex., como parte do ID/IF), que é escrito com o endereço da próxima instrução ao final de cada ciclo IF.

Estágio de pipe	Instrução de desvio
IF	IF/ID.IR ← Mem[PC]; **IF/ID.NPC,PC ← (if ((IF/ID.opcode == branch) & (Regs[IF/ID.IR$_{6..10}$] op 0)) {IF/ID.NPC + sign-extended (IF/ID.IR[immediate field] <<2) else {PC+4});**
ID	ID/EX.A ← Regs[IF/ID.IR$_{6..10}$]; ID/EX.B ← Regs[IF/ID.IR$_{11..15}$]; ID/EX.IR ← IF/ID.IR; ID/EX.Imm ← (IF/ID.IR$_{16}$)^{16}##IF/ID.IR$_{16..31}$
EX	
MEM	
WB	

FIGURA C.29 Esta estrutura de pipeline revisada é baseada no original da Figura C.23.
Ela usa um somador separado, como na Figura C.28, para calcular o endereço de destino de desvio durante ID. As operações que são novas e mudaram estão em negrito. Como a adição do endereço de destino de desvio acontece durante ID, ela acontecerá para todas as instruções; a condição de desvio (`Regs[IF/ID. IR6..10] op 0`) também será feita para todas as instruções. A seleção do PC sequencial ou do PC de destino de desvio ainda ocorre durante IF, mas agora utiliza valores do estágio ID, que corresponde aos valores definidos pela instrução anterior. Essa mudança reduz a penalidade de desvios condicionais em dois ciclos: um da avaliação do destino e condição de desvio antecipadamente, e um do controle da seleção do PC no mesmo clock, em vez de no clock seguinte. Como o valor de cond é definido como 0, a menos que a instrução em ID seja um desvio tomado, o processador precisa decodificar a instrução antes do final de ID. Como o desvio é feito no final de ID, os estágios EX, MEM e WB não são usados para desvios. Uma complicação adicional surge dos saltos que possuem um offset maior do que os desvios. Podemos resolver isso usando um somador adicional, que soma o PC e os 26 bits inferiores do IR depois de deslocados à esquerda em 2 bits.

estágios separados de decodificação e busca de registrador provavelmente terá um *atraso de desvio condicional* — a extensão do hazard de controle —, que é pelo menos um ciclo de clock maior. O atraso de desvio condicional, a menos que seja tratado, se transforma em uma penalidade de desvio. Muitas CPUs mais antigas, que implementam conjuntos de instruções mais complexos, possuem atrasos de desvio de quatro ciclos de clock ou mais, e processadores grandes, com pipelines profundos, normalmente possuem penalidades de desvio de seis ou sete. Em geral, quanto mais profundo o pipeline, pior a penalidade de desvio nos ciclos de clock. Naturalmente, o efeito de desempenho relativo de uma penalidade de desvio maior depende do CPI geral do processador. Um processador de CPI baixo pode ter desvios condicionais mais dispendiosos, porque a porcentagem do desempenho do processador que será perdida com os desvios é menor.

C.4 O QUE TORNA O PIPELINING DIFÍCIL DE IMPLEMENTAR?

Agora que entendemos como detectar e resolver os hazards, podemos lidar com algumas complicações que evitamos até o momento. A primeira parte desta seção considera os desafios das situações excepcionais cuja ordem de execução da instrução é alterada de maneiras inesperadas. Na segunda parte, discutiremos alguns dos desafios levantados por diferentes conjuntos de instruções.

Tratando de exceções

As situações excepcionais são mais complicadas de tratar em uma CPU com pipeline, pois a sobreposição de instruções dificulta saber se uma instrução pode mudar com segurança o estado da CPU. Em uma CPI em pipeline, uma instrução é executada parte por parte, e não é concluída por vários ciclos de clock. Infelizmente, outras instruções no pipeline podem levantar exceções que podem forçar a CPU a abortar as instruções no pipeline

antes que elas sejam concluídas. Antes de discutirmos esses problemas e suas soluções com detalhes, precisamos entender que tipos de situações podem surgir e que requisitos de arquitetura existem para dar suporte a elas.

Tipos de exceções e requisitos

A terminologia usada para descrever situações excepcionais em que a ordem de execução normal da instrução é alterada varia entre as CPUs. Os termos *interrupção, falta* e *exceção* são utilizados, embora não de maneira coerente. Usamos o termo *exceção* para abranger todos esses mecanismos, incluindo os seguintes:

- Solicitação de dispositivo de E/S
- Chamada de um serviço do sistema operacional a partir de um programa do usuário
- Tracing da execução da instrução
- Breakpoint (interrupção solicitada pelo programador)
- Overflow aritmético de inteiros
- Anomalia de aritmética de ponto flutuante
- Falta de página (não na memória principal)
- Acessos desalinhados à memória (se o alinhamento for exigido)
- Violação de proteção de memória
- Uso de uma instrução não definida ou não implementada
- Defeitos do hardware
- Falta de energia

Quando quisermos nos referir a alguma classe em particular de tais exceções, usaremos um nome mais longo, como interrupção de E/S, exceção de ponto flutuante ou falta de página. A Figura C.30 mostra a variedade de nomes diferentes para os eventos de exceção comuns, mostrados anteriormente.

Embora usemos o termo *exceção* para abranger todos esses eventos, eventos individuais possuem características importantes que determinam que ação é necessária no hardware. Os requisitos nas exceções podem ser caracterizados em cinco eixos semi-independentes:

1. *Síncrono* versus *assíncrono.* Se o evento ocorre no mesmo lugar toda vez que o programa é executado com os mesmos dados e alocação de memória, ele é *síncrono*. Com a exceção dos defeitos de hardware, os eventos *assíncronos* são causados por dispositivos externos à CPU e à memória. Os eventos assíncronos normalmente podem ser tratados após o término da instrução atual, que os torna mais fáceis de lidar.
2. *Solicitado pelo usuário* versus *forçado.* Se a tarefa do usuário o exigir, esse é um evento *solicitado pelo usuário*. De certa forma, as exceções solicitadas pelo usuário não são realmente exceções, pois são previsíveis. Porém, elas são tratadas como exceções, porque os mesmos mecanismos usados para salvar e restaurar o estado são usados para os eventos solicitados pelo usuário. Como a única função de uma instrução que dispara essa exceção é causar uma exceção, as exceções solicitadas pelo usuário sempre podem ser tratadas após a instrução terminar. As exceções *forçadas* são causadas por algum evento de hardware que não está sob o controle do programa do usuário. As exceções forçadas são mais difíceis de implementar porque não são previsíveis.
3. *Mascarável pelo usuário ou não mascarável.* Se um evento puder ser mascarado ou desativado por uma tarefa do usuário, ele será *mascarável pelo usuário.* Essa máscara simplesmente controla se o hardware responde à exceção ou não.
4. *Dentro ou entre instruções.* Essa classificação depende de o evento impedir que o término da instrução ocorra no meio da execução — não importa o tempo —

Evento de exceção	IBM 360	VAX	Motorola 680x0	Intel 80x86
Solicitação de dispositivo de E/S	Interrupção de entrada/saída	Interrupção de dispositivo	Exceção (nível 0...7 autovector)	Interrupção vetorada
Chamada de um serviço do sistema operacional a partir de um programa do usuário	Interrupção de chamada de supervisor	Exceção (trap altera modo do supervisor)	Exceção (instrução não implementada) no Macintosh	Interrupção (instrução INT)
Tracing da execução da instrução	Não se aplica	Exceção (falta de trace)	Exceção (trace)	Interrupção (single step trap)
Breakpoint	Não se aplica	Exceção (falta de breakpoint)	Exceção (instrução ou breakpoint ilegal)	Interrupção (trap de breakpoint)
Overflow ou underflow aritmético de inteiro; trap de ponto flutuante	Interrupção do programa (exceção de overflow ou underflow)	Exceção (trap de overflow de inteiro ou falta de underflow de ponto flutuante)	Exceção (erros de coprocessador de ponto flutuante)	Interrupção (trap de overflow ou exceção de unidade matemática)
Falta de página (não na memória principal)	Não se aplica (apenas no 370)	Exceção (falta de tradução inválida)	Exceção (erros na unidade de gerenciamento de memória)	Interrupção (falta de página)
Acessos desalinhados à memória	Interrupção do programa (exceção de especificação)	Não se aplica	Exceção (erro de endereço)	Não se aplica
Violações de proteção da memória	Interrupção do programa (exceção de proteção)	Exceção (falta de violação do controle de acesso)	Exceção (erro de barramento)	Interrupção (exceção de proteção)
Uso de instruções indefinidas	Interrupção do programa (exceção de operação)	Exceção (falta de opcode privilegiada/reservada)	Exceção (instrução ou breakpoint ilegal/instrução não implementada)	Interrupção (opcode inválido)
Defeitos de hardware	Interrupção de verificação de máquina	Exceção (aborto de verificação de máquina)	Exceção (erro de barramento)	Não se aplica
Falta de energia	Interrupção de verificação de máquina	Interrupção urgente	Não se aplica	Interrupção não mascarável

FIGURA C.30 **Os nomes das exceções comuns variam entre quatro arquiteturas diferentes.**
Cada evento no IBM 360 e no 80x86 é chamado `interrupção`, enquanto cada evento no 680 $\times$ 0 é chamado `exceção`. O VAX divide os eventos em `interrupções` ou `exceções`. Os termos *dispositivo*, *software* e *urgente* são usados com interrupções do VAX, enquanto as exceções do VAX são subdivididas em *faltas*, *traps* e *abortos*.

ou se ele é reconhecido *entre* as instruções. As exceções que ocorrem *dentro* das instruções normalmente são síncronas, pois a instrução dispara a exceção. É mais difícil implementar exceções que ocorrem dentro das instruções do que aquelas entre as instruções, pois a instrução precisa ser terminada e reiniciada. As exceções assíncronas que ocorrem dentro das instruções surgem de situações catastróficas (p. ex., defeito de hardware) e sempre causam o término do programa.

5. *Retomar ou terminar.* Se a execução do programa sempre termina após a interrupção, esse é um evento de *término*. Se a execução do programa continua após a interrupção, esse é um evento de *retomada*. É mais fácil implementar exceções que terminam a execução, pois a CPU não precisa ser capaz de reiniciar a execução do mesmo programa depois de tratar da exceção.

A Figura C.31 classifica os exemplos da Figura C.30 de acordo com essas cinco categorias. A tarefa difícil é implementar as interrupções que ocorrem dentro das instruções, quando a instrução precisa ser retomada. A implementação de tais exceções exige que outro programa

Tipo de exceção	Síncrona ou assíncrona	Solicitação do usuário ou forçado	Mascarável pelo usuário ou não mascarável	Dentro ou entre instruções	Retomar ou terminar
Solicitação de dispositivo de E/S	Assíncrona	Coagido	Não mascarável	Entre	Retomar
Invocar serviços do sistema operacional pelo programa de usuário	Síncrona	Solicitação do usuário	Não mascarável	Entre	Retomar
Tracing de execução de instrução	Síncrona	Solicitação do usuário	Mascarável pelo usuário	Entre	Retomar
Breakpoint	Síncrona	Solicitação do usuário	Mascarável pelo usuário	Entre	Retomar
Overflow aritmético de inteiro	Síncrona	Forçado	Mascarável pelo usuário	Dentro	Retomar
Overflow ou underflow aritmético de ponto flutuante	Síncrona	Forçado	Mascarável pelo usuário	Dentro	Retomar
Falta de página	Síncrona	Forçado	Não mascarável	Dentro	Retomar
Acessos à memória desalinhados	Síncrona	Forçado	Mascarável pelo usuário	Dentro	Retomar
Violações de proteção de memória	Síncrona	Forçado	Não mascarável	Dentro	Retomar
Uso de instruções indefinidas	Síncrona	Forçado	Não mascarável	Dentro	Terminar
Defeitos do hardware	Assíncrona	Forçado	Não mascarável	Dentro	Terminar
Falta de energia	Assíncrona	Forçado	Não mascarável	Dentro	Terminar

FIGURA C.31 **Cinco categorias são usadas para definir que ações são necessárias para os diferentes tipos de exceção mostrados na Figura C.30.**

As exceções que devem permitir a retomada são marcadas como retomar, embora o software normalmente possa decidir terminar o programa. Exceções síncronas e forçadas ocorrendo dentro de instruções que podem ser retomadas são as mais difíceis de implementar. Poderíamos esperar que as violações de acesso de proteção de memória sempre resultem em término; porém, os sistemas operacionais modernos utilizam proteção de memória para detectar eventos como a primeira tentativa de usar uma página ou a primeira escrita em uma página. Assim, as CPUs precisam ser capazes de retomar após essas exceções.

seja invocado para salvar o estado do programa em execução, corrigir a causa da exceção e depois restaurar o estado do programa antes que a instrução que causou a exceção possa ser tentada novamente. Esse processo precisa ser efetivamente invisível ao programa em execução. Se um pipeline oferece a capacidade para o processador tratar da exceção, salvar o estado e reiniciar sem afetar a execução do programa, o pipeline ou o processador é considerado *reiniciável*. Embora os primeiros supercomputadores e microprocessadores geralmente não tivessem essa propriedade, quase todos os processadores de hoje possuem suporte, pelo menos para o pipeline de inteiros, pois isso é necessário para implementar a memória virtual (Cap. 2).

Terminando e reiniciando a execução

Assim como nas implementações sem pipeline, as exceções mais difíceis possuem duas propriedades: 1) elas ocorrem dentro das instruções (ou seja, no meio da execução da instrução correspondente a estágios de pipe EX ou MEM); e 2) precisam ser reiniciáveis. Em nosso pipeline MIPS, por exemplo, uma falta de página de memória virtual resultante de uma busca de dados não pode ocorrer até algum tempo no estágio MEM da instrução. Quando essa falta for vista, várias outras instruções estarão em execução. Uma falta de página precisa ser reiniciável e requer a intervenção de outro processo, como o sistema operacional. Assim, o pipeline precisa ser desligado seguramente e o estado salvo de modo que a instrução possa ser reiniciada no estado correto. O reinício normalmente é

implementado salvando o PC da instrução que será reiniciada. Se a instrução reiniciada não for um desvio, continuaremos a buscar os sucessores sequenciais e iniciar sua execução no modo normal. Se a instrução reiniciada for um desvio, reavaliaremos a condição de desvio e começaremos a buscar a partir do destino ou do fall-through. Quando ocorre uma exceção, o controle do pipeline pode tomar os seguintes passos para salvar o estado do pipeline com segurança:

1. Forçar uma instrução de trap no pipeline, no próximo IF.
2. Até que o trap seja tomado, desativar todas as escritas para a instrução que falhou e para todas as instruções seguintes no pipeline; isso pode ser feito colocando-se zeros nos latches do pipeline de todas as instruções no pipeline, começando com a instrução que gera a exceção, mas não aquelas que precedem essa instrução. Isso impede quaisquer mudanças de estado para as instruções que não serão concluídas antes que a exceção seja tratada.
3. Depois que a rotina de tratamento de exceção no sistema operacional recebe o controle, ela imediatamente salva o PC da instrução que falhou. Esse valor será usado para retornar da exceção mais tarde.

Quando usamos desvios condicionais adiados, conforme mencionamos na seção anterior, não é mais possível recriar o estado do processador com um único PC, pois as instruções no pipeline podem não estar relacionadas sequencialmente. Assim, precisamos salvar e restaurar tantos PCs quanto a extensão do atraso de desvio condicional mais um. Isso é feito no terceiro passo, mostrado anteriormente.

Depois que a exceção tiver sido tratada, instruções especiais retornam o processador da exceção recarregando os PCs e reiniciando o fluxo de instruções (usando a instrução RFE no MIPS). Se o pipeline puder ser interrompido de modo que as instruções imediatamente antes da interface com falha sejam completadas e as que vêm depois possam ser reiniciadas do zero, diz-se que o pipeline possui *exceções precisas.* O ideal é que a instrução que falha não tenha mudado de estado, e o tratamento correto de algumas exceções exige que a instrução que falha não tenha efeitos. Para outras exceções, como as exceções de ponto flutuante, a instrução que falha em alguns processadores escreve seu resultado antes que a exceção possa ser tratada. Nesses casos, o hardware precisa ser preparado para apanhar os operandos-fonte, mesmo que o destino seja idêntico a um dos operandos-fonte. Como as operações de ponto flutuante podem ser executadas por muitos ciclos, é altamente provável que alguma outra instrução possa ter escrito os operandos-fonte (conforme veremos na próxima seção, as operações de ponto flutuante normalmente são completadas fora de ordem). Para contornar isso, muitas CPUs de alto desempenho recentes introduziram dois modos de operação. Um modo possui exceções precisas, e o outro (modo rápido ou de desempenho), não. Naturalmente, o modo de exceções precisas é mais lento, pois permite menos sobreposição entre as instruções de ponto flutuante. Em algumas CPUs de alto desempenho, incluindo Alpha 21064, Power2 e MIPS R8000, o modo de exceções precisas normalmente é muito mais lento (>10 vezes) e, assim, útil apenas para a depuração dos códigos.

O suporte a exceções precisas é um requisito em muitos sistemas, enquanto em outros é "simplesmente" valioso, pois simplifica a interface com sistema operacional. No mínimo, qualquer processador com paginação por demanda ou manipuladores de trap aritmético IEEE precisa tornar suas exceções precisas, seja no hardware ou com algum suporte do software. Para pipelines de inteiros, a tarefa de criar exceções precisas é mais fácil, e a acomodação da memória virtual motiva fortemente o suporte para exceções precisas para referências à memória. Na prática, esses motivos têm levado os projetistas e arquitetos a sempre oferecer exceções precisas para o pipeline de inteiros. Nesta seção, descreveremos

como implementar essas exceções no MIPS. Além disso, descreveremos as técnicas para tratar dos desafios mais complexos que surgem do pipeline de PF na Seção C.5.

Exceções no MIPS

A Figura C.32 mostra os estágios do pipeline do MIPS e quais exceções "problemáticas" poderiam ocorrer em cada estágio. Com o pipelining, múltiplas exceções podem ocorrer no mesmo ciclo de clock, pois existem múltiplas instruções em execução. Por exemplo, considere esta sequência de instruções:

LD	IF	ID	EX	MEM	WB	
DADD		IF	ID	EX	MEM	WB

Estágio do pipeline	Exceções problemáticas ocorrendo
IF	Falta de página na busca de instruções; acesso desalinhado à memória; violação de proteção de memória
ID	Opcode indefinido ou ilegal
EX	Exceção aritmética
MEM	Falta de página na busca de dados; acesso desalinhado à memória; violação de proteção de memória
WB	Nenhuma

FIGURA C.32 Exceções que podem ocorrer no pipeline do MIPS.
As exceções levantadas da instrução ou do acesso à memória de dados são responsáveis por seis dos oito casos.

Esse par de instruções pode causar uma falta de página de dados e uma exceção aritmética ao mesmo tempo, pois o LD está no estágio MEM enquanto o DADD está no estágio EX. Esse caso pode ser tratado lidando-se apenas com a falta de página de dados e depois reiniciando a execução. A segunda exceção ocorrerá novamente (mas não a primeira, se o software estiver correto), e, quando isso acontecer, ela poderá ser tratada de modo independente.

Na realidade, a situação não é tão fácil quanto nesse exemplo simples. As exceções podem ocorrer fora de ordem, ou seja, uma instrução pode causar uma exceção antes que uma instrução anterior cause uma. Considere novamente a sequência de instruções anterior, LD seguido por DADD. O LD pode gerar uma falta de página de dados, vista quando a instrução está em MEM, e o DADD pode gerar uma falta de página de instrução, vista quando a instrução DADD está em IF. A falta de página de instrução realmente ocorrerá primeiro, embora seja causada por uma instrução posterior!

Como estamos implementando exceções precisas, o pipeline precisa tratar da exceção causada pela instrução LD primeiro. Para explicar como isso funciona, vamos chamar a instrução na posição da instrução LD de i e a instrução na posição da instrução DADD de $i + 1$. O pipeline não pode simplesmente lidar com uma exceção quando ela ocorre no tempo, pois isso levará a exceções ocorrendo fora da ordem sem pipeline. Em vez disso, o hardware posta todas as exceções causadas por determinada instrução em um vetor de estado associado a essa instrução. O vetor de estado de exceção é transportado quando a instrução desce pelo pipeline. Quando uma indicação de exceção é definida no vetor de estado da exceção, qualquer sinal de controle que possa fazer com que um valor de dados seja escrito é desativado (isso inclui tanto escritas de registrador quanto escritas de

memória). Como um store pode causar uma exceção durante MEM, o hardware precisa ser preparado para impedir que o store complete, se levantar uma exceção.

Quando uma instrução entra em WB (ou está para sair de MEM), o vetor de estado da exceção é verificado. Se quaisquer exceções forem postadas, elas serão tratadas na ordem em que ocorreriam no tempo, em um processador sem pipeline — a exceção correspondente à instrução mais antiga (e normalmente o estágio de pipe mais antigo para essa instrução) é tratada primeiro. Isso garante que todas as exceções serão vistas na instrução *i* antes que quaisquer sejam vistas em $i + 1$. Naturalmente, qualquer ação tomada nos estágios de pipe anteriores em favor da instrução *i* pode ser inválida, pois como as escritas no banco de registradores e memória foram desativadas nenhum estado poderia ter sido alterado. Conforme veremos na Seção C.5, a manutenção desse modelo preciso para operações de ponto flutuante é muito mais difícil.

Na próxima subseção, descreveremos os problemas que surgem na implementação de exceções nas pipelines de processadores com instruções mais poderosas, de mais longa duração.

Complicações do conjunto de instruções

Nenhuma instrução do MIPS possui mais de um resultado, e nosso pipeline do MIPS escreve esse resultado apenas no final da execução de uma instrução. Quando uma instrução tem garantias de que completará, ela é chamada *confirmada*. No pipeline de inteiros do MIPS, todas as instruções são confirmadas quando alcançam o final do estágio MEM (ou início do WB) e nenhuma instrução atualiza o estado antes desse estágio. Assim, exceções precisas são simples. Alguns processadores possuem instruções que mudam o estado no meio da execução da instrução, antes que a instrução e seus predecessores tenham garantias de término. Por exemplo, os modos de endereçamento de autoincremento na arquitetura IA-32 causam a atualização dos registradores no meio da execução de uma instrução. Nesse caso, se a instrução for abortada devido a uma exceção, ela deixará o estado do processador alterado. Embora saibamos qual instrução causou a exceção, sem suporte adicional do hardware a exceção será imprecisa, pois a instrução será terminada pela metade. É difícil reiniciar o fluxo de instruções depois de tal exceção imprecisa. Como alternativa, poderíamos evitar a atualização do estado antes que a instrução seja confirmada, mas isso pode ser difícil e dispendioso, pois pode haver dependências no estado atualizado. Considere uma instrução do VAX que autoincrementa o mesmo registrador várias vezes. Assim, para manter um modelo de exceção preciso, a maioria dos processadores com tais instruções tem a capacidade de desfazer quaisquer mudanças de estado feitas antes que a instrução seja confirmada. Se houver uma exceção, o processador usará essa capacidade para reiniciar o estado do processador no seu valor antes que a instrução interrompida seja iniciada. Na próxima seção, veremos que um pipeline MIPS de ponto flutuante mais poderoso pode introduzir problemas semelhantes, e a Seção C.7 introduz técnicas que complicam substancialmente o tratamento da exceção.

Uma fonte de dificuldades relacionadas surge de instruções que atualizam o estado da memória durante a execução, como as operações de cópia de string no VAX ou IBM 360 (Apêndice K). Para permitir a interrupção e o reinício dessas instruções, as instruções são definidas para usar registradores de uso geral como registradores de trabalho. Assim, o estado da instrução parcialmente completada está sempre nos registradores, que são salvos em uma exceção e restaurados após a exceção, permitindo que a instrução continue. No VAX, um bit de estado adicional registra quando uma instrução iniciou a atualização do estado da memória, de modo que, quando o pipeline é reiniciado, a CPU sabe se deve

reiniciar a instrução do início ou do meio da instrução. As instruções de string IA-32 também usam os registradores como armazenamento de trabalho, de modo que salvar e restaurar os registradores salva e restaura o estado dessas instruções.

Um conjunto de dificuldades diferente surge dos bits de estado ímpares que podem criar hazards adicionais de pipeline ou exigir hardware extra para salvar e restaurar. Os códigos de condição são um bom exemplo disso. Muitos processadores definem os códigos de condição implicitamente como parte da instrução. Essa técnica tem vantagens, pois os códigos de condição desacoplam a avaliação da condição do desvio real. Porém, códigos de condição definidos implicitamente podem causar dificuldades no escalonamento de quaisquer atrasos de pipeline entre o estabelecimento do código de condição e o desvio, pois a maior parte das instruções define o código de condição e não pode ser usada nos slots de atraso entre a avaliação da condição e o desvio.

Além do mais, nos processadores com códigos de condição, o processador precisa decidir quando a condição de desvio é fixada. Isso envolve descobrir quando o código de condição foi definido pela última vez antes do desvio. Na maioria dos processadores com códigos de condição definidos implicitamente, isso é feito adiando-se a avaliação da condição de desvio até que todas as instruções anteriores tenham tido uma chance de definir o código de condição.

Naturalmente, as arquiteturas com códigos de condição definidos explicitamente permitem que o atraso entre o teste da condição e o desvio sejam programados; porém, o controle do pipeline ainda precisa rastrear a última instrução que define o código de condição para saber quando a condição de desvio é decidida. Com efeito, o código de condição precisa ser tratado como um operando que exige detecção de hazard para os hazards RAW com desvios condicionais, assim como o MIPS precisa fazer nos registradores.

Uma última área difícil no pipelining são as operações multiciclos. Imagine tentar canalizar uma sequência de instruções VAX como esta:

```
MOVL    R1,R2                  ;movimentação entre registradores
ADDL3   42(R1),56(R1)+,@(R1)   ;acrescenta locais de memória
SUBL2   R2,R3                  ;subtrai registradores
MOVC3   @(R1)[R2],74(R2),R3    ;move uma string de caracteres
```

Essas instruções diferem radicalmente no número de ciclos de clock que exigirão, desde um até centenas de ciclos de clock. Elas também exigem diferentes números de acessos à memória de dados, de zero até possivelmente centenas. Os hazards de dados são muito complexos e ocorrem entre e dentro das instruções. A solução simples de fazer com que todas as instruções sejam executadas pelo mesmo número de ciclos de clock é inaceitável, pois introduz grande número de hazards e condições de bypass e cria um pipeline imensamente longo. O pipelining do VAX no nível de instrução é difícil, mas uma solução inteligente foi descoberta pelos projetistas do VAX 8800. Eles implementarão um pipeline para a execução da *microinstrução*: uma microinstrução é uma instrução simples, usada nas sequências para implementar um conjunto de instruções mais complexas. Como as microinstruções são simples (elas se parecem muito com MIPS), o controle do pipeline é muito mais fácil. Desde 1995, todos os microprocessadores IA-32 da Intel têm utilizado essa estratégia de converter as instruções IA-32 para micro-operações e depois canalizar as micro-operações.

Em comparação, processadores load-store possuem operações simples com quantidades semelhantes de trabalho e implementadas mais facilmente com pipeline. Se os arquitetos realizarem o relacionamento entre o projeto do conjunto de instruções e o pipelining, poderão projetar arquiteturas para o pipelining mais eficiente. Na próxima seção, veremos como o pipeline MIPS lida com instruções de longa duração, especificamente operações de ponto flutuante.

Durante muitos anos, acreditou-se que a interação entre os conjuntos de instruções e as implementações era pequena, e as questões de implementação não eram o foco principal no projeto de conjuntos de instruções. Nos anos 1980, ficou claro que a dificuldade e a ineficiência do pipelining poderiam ser aumentadas pelas complicações do conjunto de instruções. Nos anos 1990, todas as empresas passaram para conjuntos de instruções mais simples, com o objetivo de reduzir a complexidade de implementações agressivas.

C.5 ESTENDENDO O PIPELINE MIPS PARA LIDAR COM OPERAÇÕES MULTICICLOS

Agora queremos explorar como nosso pipeline MIPS pode ser estendido para lidar com operações de ponto flutuante. Esta seção se concentra na técnica básica e nas alternativas de projeto, fechando com algumas medidas de desempenho de um pipeline de ponto flutuante MIPS.

É pouco prático exigir que todas as operações de ponto flutuante MIPS concluam em um ciclo de clock ou ainda em dois. Isso significaria aceitar um clock lento ou usar enorme quantidade de lógica nas unidades de ponto flutuante ou ambos.

Em vez disso, o pipeline de ponto flutuante permitirá uma latência maior para as operações. Isso é mais fácil de entender se imaginarmos as instruções de ponto flutuante como tendo a mesmo pipeline das instruções de inteiros, com duas mudanças importantes: 1) o ciclo EX pode ser repetido tantas vezes quantas forem necessárias para concluir a operação — o número de repetições pode variar para operações diferentes; 2) pode haver múltiplas unidades funcionais de ponto flutuante. Haverá um stall se a instrução a ser despachada causar um hazard estrutural para a unidade funcional que ela utiliza ou se ela causar um hazard de dados.

Para esta seção, vamos considerar que existam quatro unidades funcionais separadas em nossa implementação MIPS:

1. A unidade de inteiros principal que trata de loads e stores, operações de ALU com inteiros e desvios.
2. Multiplicador de ponto flutuante e inteiros.
3. Somador de ponto flutuante que trata de adição, subtração e conversão de PF.
4. Divisor de ponto flutuante e inteiros.

Se também considerarmos que os estágios de execução dessas unidades funcionais não estão em pipeline, então a Figura C.33 mostra a estrutura de pipeline resultante. Como EX não está em pipeline, nenhuma outra instrução usando essa unidade funcional poderá ser despachada até que a instrução anterior saia do EX. Além do mais, se uma instrução não puder prosseguir para o estágio EX, o pipeline inteiro por trás dessa instrução será adiada.

Na realidade, os resultados intermediários provavelmente não realizam um ciclo em torno da unidade EX, como sugere a Figura C.33; em vez disso, o estágio EX do pipeline possui atrasos de clock a mais do que 1. Podemos generalizar a estrutura do pipeline de PF mostrada na Figura C.33 para permitir o pipelining de alguns estágios e múltiplas operações em andamento. Para descrever tal pipeline, temos que definir tanto a latência das unidades funcionais quanto o *intervalo de iniciação* ou *intervalo de repetição*. Definimos latência como

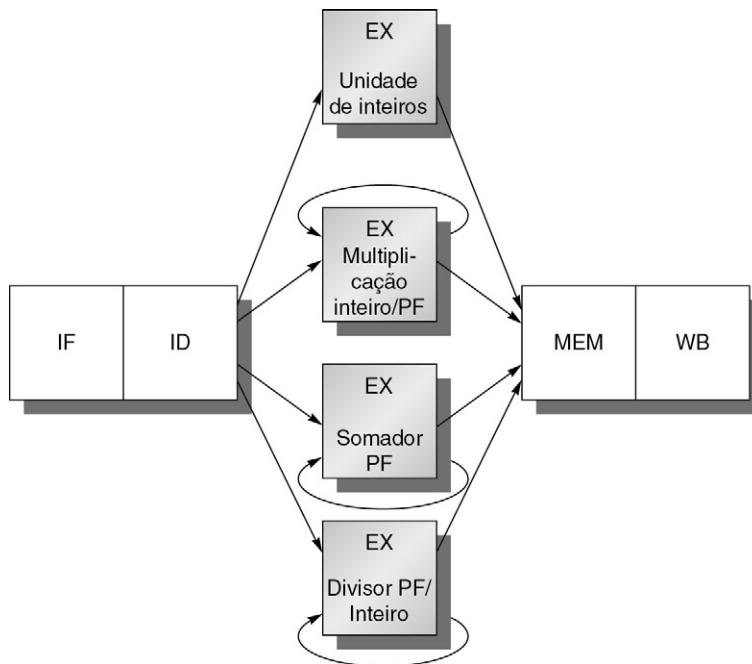



FIGURA C.33 **O pipeline MIPS com três unidades funcionais de ponto flutuante adicionais, em não pipeline.** Como somente uma instrução é despachada em cada ciclo de clock, todas as instruções passam pelo pipeline-padrão para as operações com inteiros. As operações de ponto flutuante simplesmente fazem o loop quando atingem o estágio EX. Depois de terminarem o estágio EX, elas prosseguem para MEM e WB, para concluir a execução.

fizemos anteriormente: o número de ciclos intercalados entre uma instrução que produz um resultado e uma instrução que usa o resultado. O intervalo de iniciação ou de repetição é o número de ciclos que precisam passar entre o despacho de duas operações de determinado tipo. Por exemplo, usaremos as latências e os intervalos de iniciação mostrados na Figura C.34.

Com essa definição de latência, as operações da ALU com inteiros possuem uma latência 0, pois os resultados podem ser usados no próximo ciclo de clock, e os loads têm uma latência de 1, pois seus resultados podem ser usados após um ciclo intermediário. Como a maioria das operações consome seus operandos no início de EX, a latência normalmente é o número de estágios após o EX em que uma instrução produz um resultado — por exemplo, zero estágio para operações da ALU e um estágio para loads. A principal exceção são os stores, que consomem o valor sendo armazenado um ciclo depois. Daí a latência para um store, para o valor sendo armazenado, mas não para o registrador de endereço básico, ser um ciclo a menos. A latência do pipeline é basicamente igual a um ciclo a menos que a profundidade do pipeline de execução, que é o número de estágios do estágio EX até o estágio que produz o resultado. Assim, para o pipeline de exemplo anterior,

Unidade funcional	Latência	Intervalo de iniciação
ALU de inteiros	0	1
Memória de dados (loads de inteiros e PF)	1	1
Adição de PF	3	1
Multiplicação de PF (também multiplicação de inteiros)	6	1
Divisão de PF (também divisão de inteiros)	24	25

FIGURA C.34 **Latências e intervalos de iniciação para as unidades funcionais.**

o número de estágios em uma adição de PF é quatro, enquanto o número de estágios em uma multiplicação de PF é sete. Para conseguir uma taxa de clock mais alta, os projetistas precisam colocar menos níveis lógicos em cada estágio de pipe, o que aumenta o número de estágios de pipe exigidos para operações mais complexas. A penalidade para a taxa de clock mais rápida é, portanto, latência maior para as operações.

A estrutura de pipeline de exemplo na Figura C.34 permite até quatro adições de PF pendentes, sete multiplicações de PF/inteiros pendentes e uma divisão de PF. A Figura C.35 mostra como esse pipeline pode ser projetado, estendendo a Figura C.33. O intervalo de repetição é implementado na Figura C.35 pela inclusão de estágios de pipeline adicionais, que serão separados por registradores de pipeline adicionais. Como as unidades são independentes, nomeamos os estágios de forma diferente. Os estágios de pipeline que utilizam vários ciclos de clock, como a unidade de divisão, são subdivididos ainda mais para mostrar a latência desses estágios. Como não são estágios completos, somente uma operação pode estar ativa. A estrutura do pipeline também pode ser mostrada usando-se os conhecidos diagramas apresentados anteriormente, como mostra a Figura C.36 para um conjunto de operações de PF independentes e loads e stores de PF. Naturalmente, a latência maior das operações de PF aumenta a frequência dos hazards RAW e stalls resultantes, conforme veremos mais adiante nesta seção.

A estrutura do pipeline na Figura C.35 requer a introdução de registradores de pipeline adicionais (p. ex., A1/A2, A2/A3, A3/A4) e a modificação das conexões com esses registradores. O registrador ID/EX precisa ser expandido para conectar o ID a EX, DIV, M1 e A1; podemos nos referir à parte do registrador associado a um dos próximos estágios com a notação ID/EX, ID/DIV, ID/M1 ou ID/A1. Os registradores de pipeline entre ID e todos os outros estágios podem ser imaginados como registradores separados logicamente e, na verdade, ser implementados como registradores separados. Como somente uma operação pode estar em um estágio de pipe de cada vez, a informação de controle pode ser associada ao registrador no início do estágio.

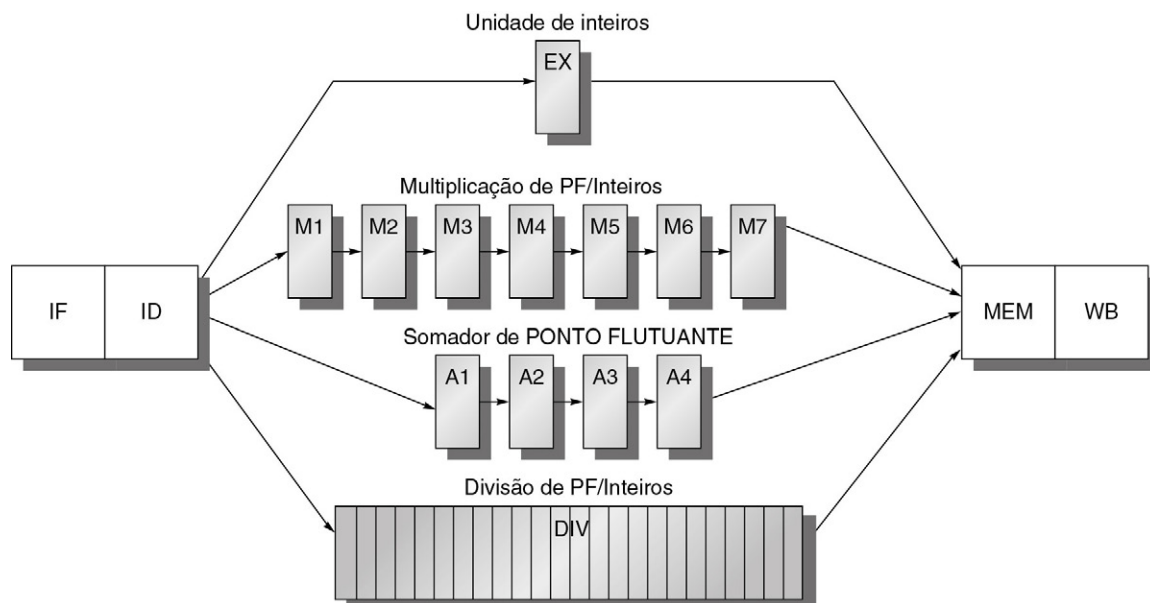



FIGURA C.35 Um pipeline que admite múltiplas operações de PF pendentes.
O multiplicador e o somador de PF estão totalmente em pipeline e possuem uma profundidade de sete e quatro estágios, respectivamente. O divisor de PF não está em pipeline, mas exige 24 ciclos de clock para concluir. A latência nas instruções entre o despacho de uma operação de PF e o uso do resultado dessa operação sem incorrer em um stall RAW é determinada pelo número de ciclos gastos nos estágios de execução. Por exemplo, a quarta instrução após uma adição de PF pode usar o resultado da adição de PF. Para operações da ALU com inteiros, a profundidade do pipeline de execução é sempre um e a próxima instrução pode usar os resultados.

MUL.D	IF	ID	M1	M2	M3	M4	M5	M6	**M7**	MEM	WB
ADD.D		IF	ID	A1	A2	A3	**A4**	**MEM**	WB		
L.D			IF	ID	EX	MEM	WB				
S.D				IF	ID	*EX*	*MEM*	WB			

FIGURA C.36 Temporização do pipeline de um conjunto de operações de PF independentes.
Os estágios em itálico mostram onde os dados são necessários, enquanto os estágios em negrito mostram onde um resultado está disponível. A extensão ".D" no mnemônico da instrução indica operações de ponto flutuante com precisão dupla (64 bits). Loads e stores de PF utilizam uma via de 64 bits para a memória, de modo que a temporização do pipelining é exatamente como um load ou store de inteiros.

Hazards e adiantamento nos pipelines de latência maior

Existem diversos aspectos diferentes na detecção de hazard e adiantamento para um pipeline como a da Figura C.35.

1. Como a unidade de divisão não está totalmente em pipeline, podem ocorrer hazards estruturais. Estes precisarão ser detectados, e o despacho das instruções terá de ser atrasado.
2. Como as instruções possuem tempos de execução variáveis, o número de escritas necessárias nos registradores em um ciclo pode ser maior do que 1.
3. Hazards WAW são possíveis, pois as instruções não atingem mais o WB em ordem. Observe que os hazards WAR não são possíveis, pois as leituras de registradores sempre ocorrem em ID.
4. As instruções podem ser completadas em uma ordem diferente das que foram despachadas, causando problemas com exceções; trataremos disso na próxima subseção.
5. Devido à latência maior das operações, os stalls para os hazards RAW serão mais frequentes.

O aumento nos stalls, que surge das latências de operação mais longas, é fundamentalmente o mesmo que é usado para o pipeline de inteiros. Antes de descrever os novos problemas que surgem nesse pipeline de PF e de procurar as soluções, vamos examinar o impacto em potencial dos hazards RAW. A Figura C.37 mostra uma sequência de código de PF típica e os stalls resultantes. Ao final desta seção, examinaremos o desempenho desse pipeline de PF para o nosso subconjunto SPEC.

Agora veja os problemas que surgem com as escritas, descritos como 2 e 3 na lista anterior. Se considerarmos que o banco de registradores de PF possui uma porta de escrita, as sequências de operações de ponto flutuante, além de um load de PF junto com operações de PF, podem causar conflitos para a porta de escrita do registrador. Considere a sequência de pipeline

	Número do ciclo de clock																
Instrução	**1**	**2**	**3**	**4**	**5**	**6**	**7**	**8**	**9**	**10**	**11**	**12**	**13**	**14**	**15**	**16**	**17**
L.D F4,0(R2)	IF	ID	EX	MEM	WB												
MUL.D F0,F4,F6		IF	ID	stall	M1	M2	M3	M4	M5	M6	M7	MEM	WB				
ADD.D F2,F0,F8			IF	stall	ID	stall	stall	stall	stall	stall	stall	A1	A2	A3	A4	MEM	WB
S.D F2,0(R2)					IF	stall	stall	stall	stall	stall	stall	ID	EX	stall	stall	stall	MEM

FIGURA C.37 Uma sequência de código de PF típica mostrando os stalls que surgem de hazards RAW.
O pipeline mais longo aumenta substancialmente a frequência de stalls *versus* o pipeline de inteiros mais superficial. Cada instrução nessa sequência é dependente da anterior e prossegue assim que os dados estão disponíveis, o que assume que o pipeline possui bypassing e adiantamento completo. O S.D precisa ser adiado por um ciclo extra, de modo que seu MEM não entre em conflito com o ADD.D. Um hardware extra poderia facilmente tratar desse caso.

Instrução	Número do ciclo de clock 1	2	3	4	5	6	7	8	9	10	11
MUL.D F0,F4,F6	IF	ID	M1	M2	M3	M4	M5	M6	M7	MEM	WB
...		IF	ID	EX	MEM	WB					
...			IF	ID	EX	MEM	WB				
ADD.D F2,F4,F6				IF	ID	A1	A2	A3	A4	MEM	WB
...					IF	ID	EX	MEM	WB		
...						IF	ID	EX	MEM	WB	
L.D F2,0(R2)							IF	ID	EX	MEM	WB

FIGURA C.38 **Três instruções desejam realizar um write-back no banco de registradores de PF simultaneamente, como mostra o ciclo de clock 11.** Este não é o pior caso, pois uma divisão anterior na unidade de PF também poderia terminar no mesmo clock. Observe que, embora MUL.D, ADD.D e L.D estejam no estágio MEM no ciclo de clock 10, somente o L.D realmente utiliza a memória, de modo que não existe um hazard estrutural para MEM.

mostrada na Figura C.38. No ciclo de clock 11, todas as três instruções alcançarão WB e desejarão escrever no banco de registradores. Com apenas uma única porta de escrita do banco de registradores, o processador precisa serializar o término da instrução. Essa única porta de registrador representa um hazard estrutural. Poderíamos aumentar o número de portas de escrita para resolver isso, mas essa solução pode não ser atraente, pois as portas de escrita adicionais seriam usadas apenas raramente. Isso porque o número máximo de portas de escrita necessárias no estado fixo é 1. Em vez disso, escolhemos detectar e forçar o acesso à porta de escrita como um hazard estrutural.

Existem duas maneiras diferentes de implementar esse interbloqueio. A primeira é rastrear o uso da porta de escrita no estágio ID e adiar uma instrução antes que ela seja despachada, assim como faríamos para qualquer outro hazard estrutural. Rastrear o uso da porta de escrita pode ser feito com um registrador de deslocamento que indica quando as instruções já despachadas usarão o banco de registradores. Se a instrução em ID precisar usar o banco de registradores ao mesmo tempo em que uma instrução já despachada, a instrução em ID será adiada por um ciclo. Em cada clock, o registrador de reserva é deslocado em 1 bit. Essa implementação tem uma vantagem: ela mantém a propriedade de que toda a detecção de interbloqueio e inserção de stall ocorre no estágio ID. O custo é o acréscimo do registrador de shift e da lógica de conflito de escrita. Assumiremos esse esquema durante toda esta seção.

Um esquema alternativo é adiar uma instrução em conflito quando ela tentar entrar no estágio MEM ou WB. Se esperarmos para adiar as instruções em conflito até que queiram entrar no estágio MEM ou WB, podemos escolher a instrução que será adiada. Uma heurística simples, embora às vezes abaixo da ideal, é dar prioridade à unidade com maior latência, pois essa é aquela que provavelmente terá causado o stall de outra instrução para um hazard RAW. A vantagem desse esquema é que ele não exige que detectemos o conflito antes da entrada do estágio MEM ou WB, onde é fácil de ser visto. A desvantagem é que isso complica o controle do pipeline, pois os stalls agora podem surgir de dois lugares. Observe que o adiamento antes de entrar em MEM fará com que o estágio EX, A4 ou M7 seja ocupado, possivelmente forçando o stall a recuar no pipeline. De modo semelhante, o stall antes de WB causaria o recuo de MEM.

Nosso outro problema é a possibilidade de hazards WAW. Para ver que eles existem, considere o exemplo da Figura C.38. Se a instrução L.D fosse despachada um ciclo antes e tivesse um destino de F2, ela criaria um hazard WAW, pois escreveria F2 um ciclo antes do ADD.D. Observe que esse hazard só ocorre quando o resultado do ADD.D é sobrescrito

sem qualquer instrução sequer o utilizar! Se houvesse um uso de F2 entre o ADD.D e o L.D, o pipeline precisaria ser atrasado para um hazard RAW, e o L.D não seria despachado até que o ADD.D fosse concluído. Argumentaríamos que, para a nosso pipeline, os hazards WAW só ocorrem quando uma instrução sem utilidade é executada, mas ainda precisamos detectá-los e nos certificar de que o resultado do L.D aparece em F2 quando terminarmos. (Conforme veremos na Seção C.8, às vezes tais sequências *ocorrem* em um código razoável.)

Existem duas maneiras possíveis de lidar com esse hazard WAW. A primeira técnica é atrasar o despacho da instrução load até que o ADD.D entre em MEM. A segunda técnica é estampar o resultado do ADD.D detectando o hazard e alterando o controle de modo que o ADD.D não escreva seu resultado. Depois, o L.D pode ser despachado imediatamente. Como esse hazard é raro, qualquer esquema funcionará bem — você pode escolher o que for mais simples de implementar. De qualquer forma, o hazard pode ser detectado durante ID quando o L.D estiver sendo emitido. Depois, será fácil atrasar o L.D ou tornar o ADD.D um no-op. A situação difícil é detectar que o L.D poderia terminar antes do ADD.D, pois isso exige saber a extensão do pipeline e a posição atual do ADD.D. Felizmente, essa sequência de código (duas escritas sem uma leitura no intervalo) será muito rara, de modo que podemos usar uma solução simples: se uma instrução em ID quiser escrever o mesmo registrador que uma instrução já despachada, não despache a instrução para EX. Na Seção C.7, veremos como o hardware adicional pode eliminar stalls para esses hazards. Primeiro, vamos juntar as partes para implementar o hazard e a lógica de despacho em nosso pipeline de PF.

Detectando os hazards possíveis, temos de considerar os hazards entre as instruções de PF, além de hazards entre uma instrução de PF e uma instrução de inteiros. Exceto por loads-stores de PF e moves de registrador de PF inteiros, os registradores de PF e inteiros são distintos. Todas as instruções com inteiros operam sobre os registradores de inteiros, enquanto as operações de ponto flutuante operam apenas sobre seus próprios registradores. Assim, só precisamos considerar os *loads-stores* de PF e os *moves* de registradores de PF na detecção de hazards entre instruções de PF e inteiros. Essa simplificação de controle de pipeline é uma vantagem adicional de se ter arquivos de registradores separados para dados inteiros e de ponto flutuante. (As principais vantagens são dobrar o número de registradores sem tornar qualquer conjunto maior, e um aumento na largura de banda sem acrescentar mais portas a qualquer conjunto. A principal desvantagem, além da necessidade de um banco de registradores extra, é o pequeno custo de *moves* ocasionais necessários entre os dois conjuntos de registradores.) Supondo que o pipeline faça toda a detecção de hazard em ID, existem três verificações que precisam ser realizadas antes que uma instrução possa ser despachada:

1. *Verificar hazards estruturais.* Espere até que a unidade funcional solicitada não esteja ocupada (isso só é necessário para divisões nesse pipeline) e certifique-se de que a porta de escrita de registrador esteja disponível quando ela for necessária.
2. *Verificar um hazard de dados RAW.* Espere até que os registradores-fonte não estejam listados como destinos pendentes em um registrador de pipeline que não estará disponível quando essa instrução precisar do resultado. Diversas verificações precisam ser feitas aqui, dependendo da instrução de origem, que determina quando o resultado estará disponível, e a instrução de destino, que determina quando o valor será necessário. Por exemplo, se a instrução em ID for uma operação de PF com registrador-fonte F2, então F2 não poderá ser listado como um destino em ID/A1, A1/A2 ou A2/A3, que corresponde à instrução de adição de PF que não serão terminadas quando a instrução em ID precisar de um resultado (ID/A1 é a parte do registrador de saída de ID que é enviada a A1). A divisão é um pouco mais complicada, se quisermos permitir que os últimos poucos ciclos de uma divisão sejam sobrepostos, pois temos de lidar com o caso em que uma divisão

está perto de terminar como especial. Na prática, os projetistas poderiam ignorar essa otimização em favor de um teste de despacho mais simples.

3. *Verificar um hazard de dados WAW.* Determine se qualquer instrução em A1, …, A4, D, M1, …, M7 tem o mesmo destino de registrador dessa instrução. Se tiver, atrase o despacho da instrução em ID.

Embora a detecção de hazard seja mais complexa com as operações de PF multiciclos, os conceitos são iguais aos do pipeline de inteiros do MIPS. O mesmo acontece para a lógica de adiantamento. O adiantamento pode ser implementado verificando se o registrador de destino em qualquer um dos registradores EX/MEM, A4/MEM, M7/MEM, D/MEM ou MEM/WB é um dos registradores-fonte de uma instrução de ponto flutuante. Se for, o multiplexador de entrada apropriado terá de ser ativado de modo a escolher os dados encaminhados. Nos exercícios, você terá a oportunidade de especificar a lógica para a detecção de hazard RAW e WAW, e também para o adiantamento.

As operações de PF multiciclos também introduzem problemas para os nossos mecanismos de exceção, de que trataremos em seguida.

Mantendo exceções precisas

Outro problema causado por essas instruções de longa duração pode ser ilustrado com a seguinte sequência de código:

```
DIV.D      F0,F2,F4
ADD.D      F10,F10,F8
SUB.D      F12,F12,F14
```

Essa sequência parece simples; existem duas dependências. Porém, surge um problema porque uma instrução despachada anteriormente pode ser completada antes de uma instrução despachada depois. Neste exemplo, podemos esperar que ADD.D e SUB.D sejam concluídas *antes* que DIV.D termine. Isso é chamado *término fora de ordem* e é comum nos pipelines com operações de longa duração (Seção C.7). Como a detecção de hazard impedirá que qualquer dependência entre as instruções seja violada, por que o término fora de ordem é um problema? Suponha que o SUB.D cause uma exceção aritmética de ponto flutuante em um ponto em que o ADD.D tiver terminado, mas o DIV.D não. O resultado será uma exceção imprecisa, algo que estamos tentando evitar. Pode parecer que isso seria tratado permitindo-se o dreno do pipeline de ponto flutuante, como fazemos para o pipeline de inteiros. Mas a exceção pode estar em uma posição onde isso não é possível. Por exemplo, se o DIV.D decidisse apanhar uma exceção aritmética de ponto flutuante após a conclusão da adição, poderíamos não ter uma exceção precisa no nível de hardware. De fato, como o ADD.D destrói um de seus operandos, não poderíamos restaurar o estado para o que era antes do DIV.D, mesmo com a ajuda do software.

Esse problema aparece porque as instruções estão completando em uma ordem diferente das que foram despachadas. Existem quatro técnicas possíveis para lidar com o término fora de ordem. A primeira é ignorar o problema e conformar-se com as exceções imprecisas. Essa técnica foi usada nos anos 1960 e no início dos anos 1970. Ela ainda é usada em alguns supercomputadores, nos quais certas classes de exceções não são permitidas ou são tratadas pelo hardware sem terminar o pipeline. É difícil usar essa técnica na maioria dos processadores embarcados hoje, devido aos recursos como memória virtual e padrão de ponto flutuante IEEE, que basicamente exigem exceções precisas por uma combinação de hardware e software. Como já dissemos, alguns processadores recentes solucionaram esse problema introduzindo dois modos de execução: um modo rápido, mas possivelmente impreciso, e um modo mais

lento, preciso. O modo preciso mais lento é implementado com uma troca de modo ou pela inserção de instruções explícitas que testam exceções de PF. De qualquer forma, a quantidade de sobreposição e reordenação permitida no pipeline de PF é significativamente restrita, de modo que efetivamente apenas uma instrução de PF está ativa de uma só vez. Essa solução é usada no DEC Alpha 21064 e 21164, no IBM Power1 e Power2, e no MIPS R8000.

Uma segunda técnica é colocar em buffer os resultados de uma operação até que todas as operações que foram despachadas anteriormente sejam concluídas. Algumas CPUs realmente utilizam essa solução, mas ela se torna dispendiosa quando a diferença nos tempos de execução entre as operações é muito grande, pois o número de resultados a colocar em buffer pode se tornar grande. Além do mais, os resultados da fila precisam ser contornados para continuar emitindo instruções enquanto se espera pela instrução maior. Isso exige um número grande de comparadores e um multiplexador muito grande.

Existem duas variações visíveis nessa técnica básica. A primeira é um *arquivo de histórico*, usado no CYBER 180/990. O arquivo de histórico registra os valores originais dos registradores. Quando ocorre uma exceção e o estado precisa ser revertido antes de alguma instrução que foi concluída fora de ordem, o valor original do registrador pode ser restaurado a partir do arquivo de histórico. Uma técnica semelhante é usada para o autoincremento e autodecremento em processadores como VAXes. Outra técnica, o *arquivo futuro*, proposta por Smith e Pleszkun (1988), mantém o valor mais novo de um registrador; quando todas as instruções anteriores tiverem sido concluídas, o banco de registradores principal é atualizado a partir do arquivo futuro. Em uma exceção, o banco de registradores principal tem os valores precisos para o estado interrompido. No Capítulo 2, vemos extensões dessa ideia, que são usadas nos processadores como o PowerPC 620 e o MIPS R10000 para permitir a sobreposição e a reordenação enquanto preserva as exceções precisas.

Uma terceira técnica em uso é permitir que as exceções se tornem um tanto imprecisas, mas manter informações suficientes para que as rotinas de tratamento de trap possam criar uma sequência precisa para a exceção. Isso significa saber quais operações estavam no pipeline e seus PCs. Depois, após tratar da exceção, o software termina quaisquer instruções anteriores à última instrução concluída, e a sequência pode ser reiniciada. Considere esta sequência de código do pior caso:

> Instrução$_1$ Uma instrução de longa duração que, por fim, interrompe a execução.
> Instrução$_2$ …, Instrução$_{n-1}$ Uma série de instruções que não estão concluídas.
> Instrução$_n$ Uma instrução que está concluída.

Dados os PCs de todas as instruções no pipeline e o PC de retorno da exceção, o software pode encontrar o estado da instrução$_1$ e da instrução$_n$. Como a instrução$_n$ foi concluída, desejaremos reiniciar a execução na instrução$_{n+1}$. Depois de tratar da exceção, o software precisa simular a execução de instrução$_2$ …, instrução$_{n-1}$. Depois, podemos retornar da exceção e reiniciar na instrução$_{n+1}$. A complexidade da execução dessas instruções corretamente pelo manipulador é a maior dificuldade desse esquema.

Existe uma simplificação importante para pipelines simples do tipo MIPS: se instrução$_2$, …, instrução$_n$ forem instruções de inteiros, saberemos que, se instrução$_n$ tiver sido concluída, todas entre instrução$_2$ …, instrução$_{n-1}$ também terão sido concluídas. Assim, somente operações de ponto flutuante precisam ser tratadas. Para tornar esse esquema tratável, o número de instruções de ponto flutuante que podem ser sobrepostas na execução pode ser limitado. Por exemplo, se somente sobrepormos duas instruções, somente a instrução interrompendo precisa ser concluída pelo software. Essa restrição pode reduzir o throughput

em potencial se os pipelines de PF forem profundos ou se houver número significativo de unidades funcionais de PF. Essa técnica é utilizada na arquitetura SPARC para permitir a sobreposição de operações de ponto flutuante e inteiros.

A técnica final é um esquema híbrido, que permite ao despacho de instruções continuar apenas se for certo que todas as instruções antes da instrução despachada terminem sem causar uma exceção. Isso garante que, quando houver uma exceção, nenhuma instrução após a que está interrompendo será concluída e todas as instruções antes da que está interrompendo podem ser concluídas. Isso, às vezes, significa atrasar a CPU para manter exceções precisas. Para que esse esquema funcione, as unidades funcionais de ponto flutuante precisam determinar se uma exceção é possível antes no estágio EX (nos três primeiros ciclos de clock no pipeline MIPS), de modo a impedir que outras instruções sejam concluídas. Esse esquema é usado no MIPS R2000/3000, no R4000 e no Intel Pentium. Isso será discutido melhor no Apêndice J.

Desempenho do pipeline de PF do MIPS

O pipeline MIPS de ponto flutuante da Figura C.35 pode gerar stalls estruturais para a unidade de divisão e stalls para hazards RAW (ela também pode ter hazards WAW, mas isso raramente ocorre na prática). A Figura C.39 mostra o número de ciclos de stall para

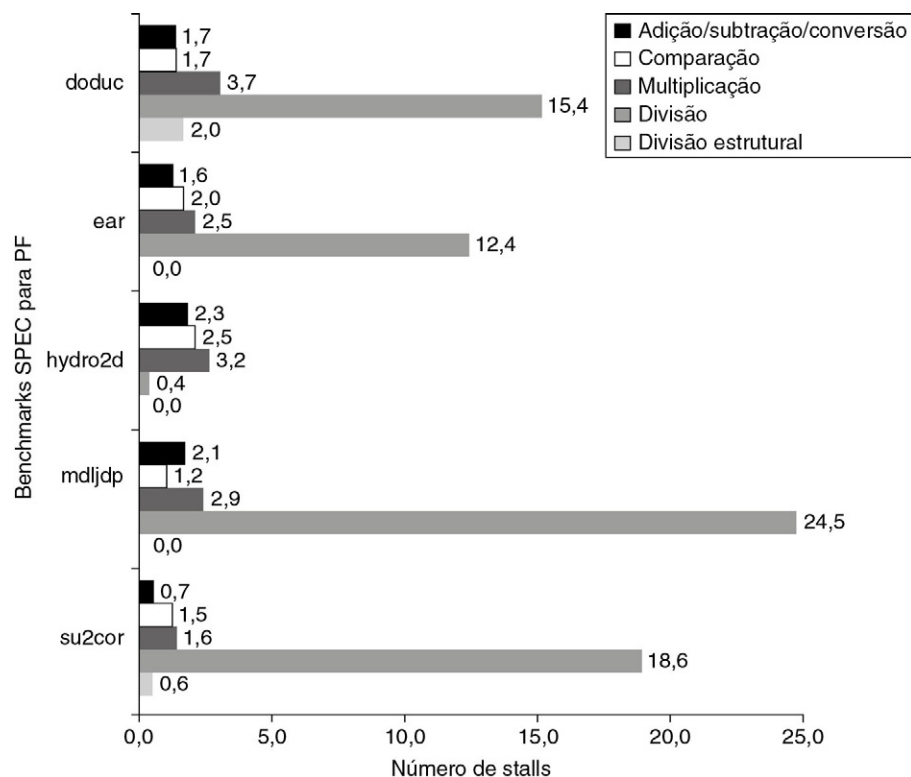



FIGURA C.39 Stalls por operação de PF para cada tipo principal da operação de PF para os benchmarks PF do SPEC89.

Exceto para os hazards estruturais de divisão, esses dados não dependem da frequência de uma operação, somente de sua latência e do número de ciclos antes que o resultado seja usado. O número de stalls dos hazards RAW rastreia aproximadamente a latência da unidade de PF. Por exemplo, o número médio de stalls por adição, subtração ou conversão de PF é de 1,7 ciclo ou 56% da latência (três ciclos). De modo semelhante, o número médio de stalls para multiplicações e divisões é de 2,8 e 14,2, respectivamente, ou 46% e 59% da latência correspondente. Os hazards estruturais para divisões são raros, pois a frequência de divisão é baixa.

cada tipo de operação de ponto flutuante com base em cada instância (ou seja, a primeira barra para cada benchmark de PF mostra o número de stalls de resultado de PF para cada adição, subtração ou conversão de PF). Como poderíamos esperar, os ciclos de stall por operação rastreiam a latência das operações de PF, variando de 46-59% da latência da unidade funcional.

A Figura C.40 oferece o desmembramento completo dos stalls de inteiros e de ponto flutuante para cinco benchmarks SPECfp. Aparecem quatro classes de stalls: stalls de resultado de PF, stalls de comparação de PF, atrasos de load e desvio condicional, e atrasos estruturais de ponto flutuante. O compilador tenta escalonar atrasos de load e PF antes de escalonar os atrasos de desvio. O número total de stalls por instrução varia de 0,65 a 1,21.

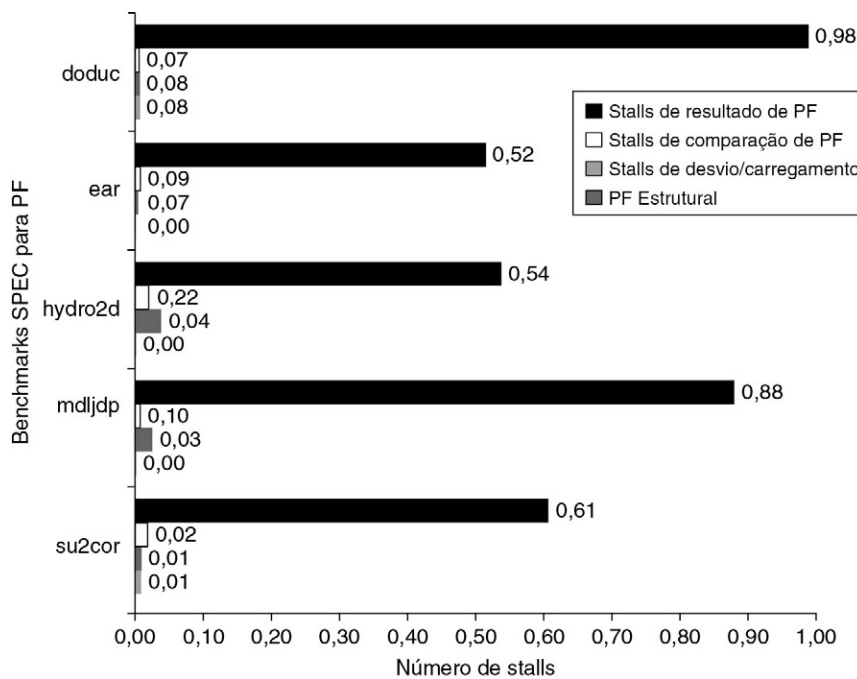



FIGURA C.40 **Stalls ocorrendo para o pipeline de PF do MIPS para cinco dos benchmarks de PF do SPEC89.**

O número total de stalls por instrução varia de 0,65 para o su2cor até 1,21 para o dudoc, com uma média de 0,87. Os stalls de resultado de PF dominam em todos os casos, com média de 0,71 stall por instrução ou 82% dos ciclos adiados. As comparações geram uma média de 0,1 stall por instrução e são a segunda maior fonte. O hazard estrutural de divisão só é significativo para o doduc.

C.6 JUNTANDO TUDO: O PIPELINE MIPS R4000

Nesta seção, vemos a estrutura do pipeline e o desempenho da família de processadores MIPS R4000, que inclui o 4400. O R4000 implementa MIPS64, mas usa um pipeline mais profundo do que o do nosso projeto de cinco estágios, para programas tanto de inteiros quanto de ponto flutuante. Esse pipeline mais profundo permite que ele atinja taxas de clock mais altas, decompondo o pipeline de inteiros de cinco estágios para oito estágios. Como o acesso à cache tem um tempo particularmente crítico, os estágios de pipeline extras vêm da decomposição do acesso à memória. Esse tipo de pipelining mais profundo às vezes é chamado *superpipelining*.

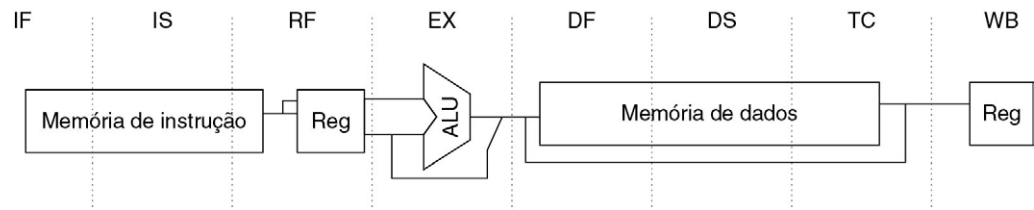



FIGURA C.41 **A estrutura de pipeline de oito estágios do R4000 usa caches de instrução e dados em pipeline.** Os estágios de pipe são rotulados e sua função detalhada é descrita no texto. As linhas verticais tracejadas representam os limites de estágio e também o local dos latches do pipeline. A instrução na realidade está disponível ao final do IS, mas a verificação de tag é feita em RF, enquanto os registradores são lidos. Assim, mostramos a memória de instruções como operando por meio do RF. O estágio TC é necessário para o acesso à memória de dados, pois não podemos escrever os dados no registrador até sabermos se o acesso à cache foi um acerto ou não.

A Figura C.41 mostra a estrutura de pipeline de oito estágios usando uma versão resumida do datapath. A Figura C.42 mostra a sobreposição de instruções bem-sucedidas no pipeline. Observe que, embora a memória de instrução e de dados ocupe vários ciclos, ela utiliza totalmente o pipeline, de modo que uma nova instrução pode iniciar a cada clock. De fato, o pipeline usa os dados antes que a detecção de acerto de cache se complete; o Capítulo 2 discute com mais detalhes como isso pode ser feito.

A função de cada estágio é a seguinte:

- IF. Primeira metade da busca de instrução; na realidade, a seleção do PC acontece aqui, junto com o início do acesso à cache de instruções.
- IS. Segunda metade da busca de instrução, completa acesso à cache de instruções.
- RF. Decodificação de instrução e busca de registrador, verificação de hazard e também detecção de acerto da cache de instruções.
- EX. Execução, que inclui cálculo do endereço efetivo, operação da ALU e avaliação de cálculo e condição de destino de desvio.
- DF. Busca de dados, primeira metade do acesso à cache de dados.
- DS. Segunda metade da busca de dados, término do acesso à cache de dados.

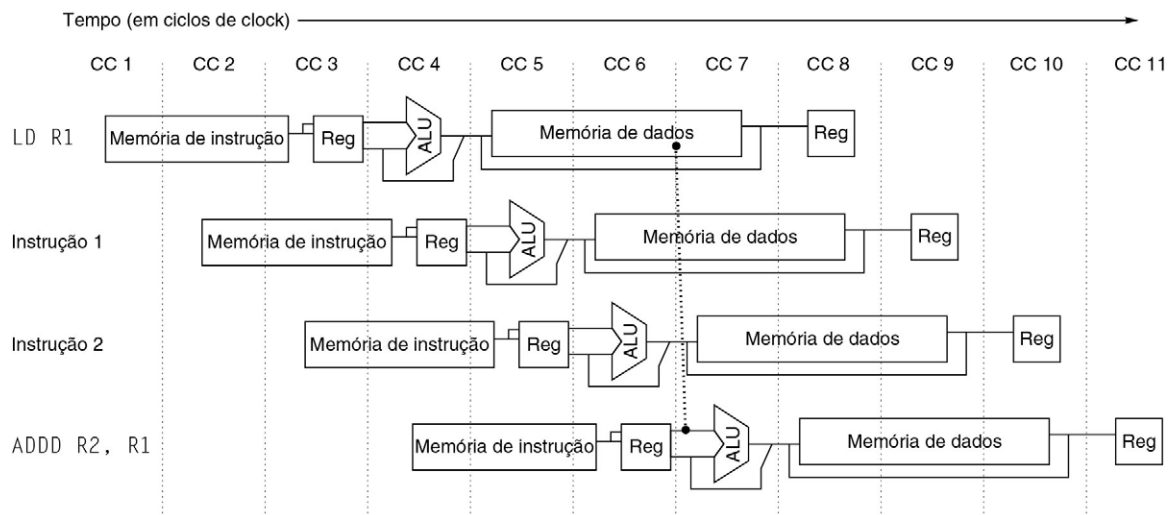



FIGURA C.42 **A estrutura do pipeline de inteiros do R4000 leva a um atraso de load de dois ciclos.** Um atraso de dois ciclos é possível porque o valor de dados está disponível ao final do DS e pode ser evitado. Se a verificação de tag no TC indicar uma falta, o pipeline é recuado por um ciclo, quando os dados corretos estarão disponíveis.

- TC. Verificação de tag determina se houve acerto no acesso à cache de dados.
- WB. Write-back para operações de load e operações registrador-registrador.

Além de aumentar substancialmente a quantidade de adiantamento exigida, esse pipeline de maior latência aumenta os atrasos tanto de load quanto de desvio condicional. A Figura C.42 mostra que os atrasos de load são de dois ciclos, pois o valor de dados está disponível no final do DS. A Figura C.43 mostra o escalonamento de pipeline resumido quando um uso é seguido imediatamente de um load. Ela mostra que o adiantamento é exigido para o resultado de uma instrução de load a um destino que está 3-4 ciclos mais adiante.

A Figura C.44 mostra que o atraso de desvio condicional básico é de três ciclos, pois a condição de destino é calculada durante EX. A arquitetura do MIPS possui um desvio adiado de único ciclo. O R4000 utiliza uma estratégia prevista não tomada para os dois ciclos restantes do atraso de desvio condicional. Como mostra a Figura C.45, os desvios não tomados são simplesmente desvios adiados de um ciclo, enquanto os desvios tomados possuem um slot de atraso de um ciclo, seguido por dois ciclos ociosos. O conjunto de instruções oferece uma instrução de desvio provável, que descrevemos anteriormente e que ajuda no preenchimento do slot de atraso de desvio condicional. Os interbloqueios de

	Número do clock								
Número da instrução	**1**	**2**	**3**	**4**	**5**	**6**	**7**	**8**	**9**
`LD R1,...`	IF	IS	RF	EX	DF	DS	TC	WB	
`DADD R2,R1,...`		IF	IS	RF	stall	stall	EX	DF	DS
`DSUB R3,R1,...`			IF	IS	stall	stall	RF	EX	DF
`OR R4,R1,...`				IF	stall	stall	IS	RF	EX

FIGURA C.43 Uma instrução de load seguida por um uso imediato resulta em um stall de dois ciclos.
As vias de adiantamento normais podem ser usadas após dois ciclos, de modo que o DADD e o DSUB recebem o valor pelo adiantamento após o stall. A instrução OR apanha o valor do banco de registradores. Como as duas instruções após o load poderiam ser independentes, e portanto não adiar, o bypass pode ser para instruções que estão 3-4 ciclos após o load.

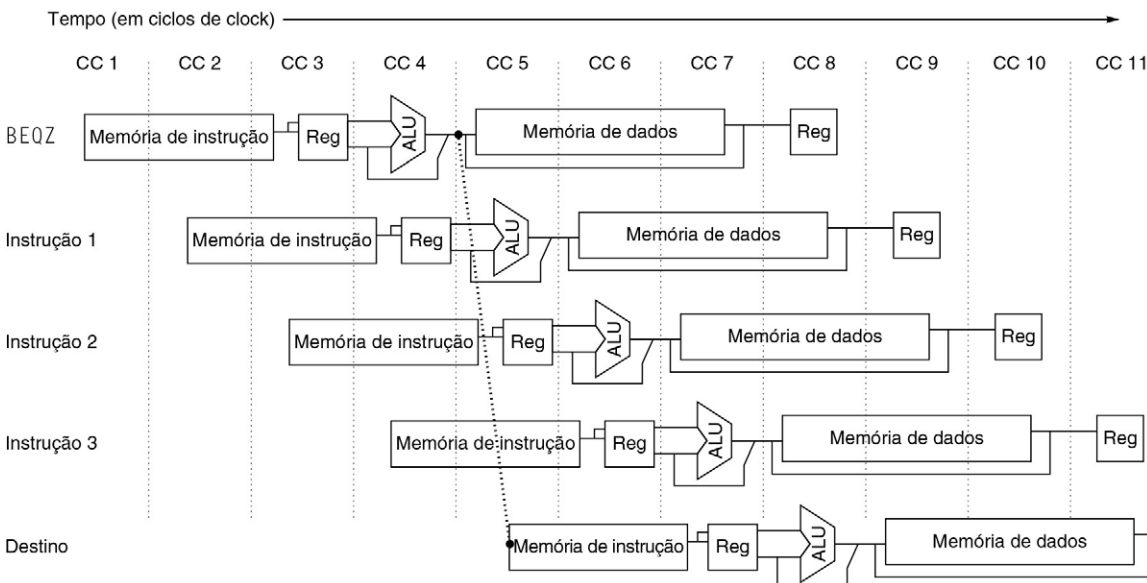



FIGURA C.44 O atraso de desvio condicional básico é de três ciclos, pois a avaliação da condição é realizada durante EX.

	Número do clock								
Número da instrução	**1**	**2**	**3**	**4**	**5**	**6**	**7**	**8**	**9**
Instrução de desvio	IF	IS	RF	EX	DF	DS	TC	WB	
Slot de atraso		IF	IS	RF	EX	DF	DS	TC	WB
Stall			stall	stall	stall	stall	stall	stall	stall
Stall				stall	stall	stall	stall	stall	stall
Destino do desvio					IF	IS	RF	EX	DF

	Número do clock								
Número da instrução	**1**	**2**	**3**	**4**	**5**	**6**	**7**	**8**	**9**
Instrução de desvio	IF	IS	RF	EX	DF	DS	TC	WB	
Slot de atraso		IF	IS	RF	EX	DF	DS	TC	WB
Instrução de desvio + 2			IF	IS	RF	EX	DF	DS	TC
Instrução de desvio + 3				IF	IS	RF	EX	DF	DS

FIGURA C.45 Um desvio tomado, mostrado na parte superior da figura, possui um slot de atraso de um ciclo seguido por um stall de dois ciclos, enquanto um desvio não tomado, mostrado na parte inferior, tem simplesmente um slot de atraso de um ciclo.
A instrução de desvio pode ser um desvio adiado comum ou um desvio provável, que cancela o efeito da instrução no slot de atraso se o desvio não for tomado.

pipeline impõem uma penalidade de stall de desvio de dois ciclos em um desvio tomado e qualquer stall de hazard de dados que surge do uso de um resultado de load.

Além do aumento nos stalls para loads e desvios, o pipeline mais profundo aumenta o número de níveis de adiantamento para as operações com a ALU. Em nosso pipeline MIPS de cinco estágios, o adiantamento entre duas instruções da ALU registrador-registrador poderia acontecer dos registradores ALU/MEM ou MEM/WB. No pipeline R4000, existem quatro fontes possíveis para o bypass da ALU: EX/DF, DF/DS, DS/TC e TC/WB.

O pipeline de ponto flutuante

A unidade de ponto flutuante R4000 consiste em três unidades funcionais: um divisor de ponto flutuante, um multiplicador de ponto flutuante e um somador de ponto flutuante. A lógica do somador é usada na etapa final de uma multiplicação ou divisão. As operações de PF de precisão dupla podem levar de dois ciclos (para uma negação) até 112 ciclos para uma raiz quadrada. Além disso, as diversas unidades possuem diferentes taxas de iniciação. A unidade funcional de ponto flutuante pode ser imaginada como tendo oito estágios diferentes, listados na Figura C.46; esses estágios são combinados em diferentes ordens para executar diversas operações de PF.

Existe uma única cópia de cada um desses estágios, e diversas instruções podem usar um estágio zero ou mais vezes e em diferentes ordens. A Figura C.47 mostra a latência, a taxa de iniciação e os estágios de pipeline usados pelas operações mais comuns de PF de precisão dupla.

Pela informação da Figura C.47, podemos determinar se uma sequência de operações de PF diferentes e independentes pode ser despachada sem stalls. Se a temporização da sequência for tal que ocorra um conflito para um estágio de pipeline compartilhado, um stall será necessário. As Figuras C.48, C.49, C.50 e C.51 mostram quatro sequências possíveis de duas instruções:

Estágio	Unidade funcional	Descrição
A	Somador de PF	Estágio de mantissa ADD
D	Divisor de PF	Estágio de divisão do pipeline
E	Multiplicador de PF	Estágio de teste de exceção
M	Multiplicador de PF	Primeiro estágio do multiplicador
N	Multiplicador de PF	Segundo estágio do multiplicador
R	Somador de PF	Estágio de arredondamento
S	Somador de PF	Estágio de deslocamento de operando
U		Desempacotamento de números de PF

FIGURA C.46 Os oito estágios usados nos pipelines de ponto flutuante do R4000.

Instrução de PF	Latência	Intervalo de iniciação	Estágios de pipe
Adição, subtração	4	3	U, S + A, A + R, R + S
Multiplicação	8	4	U, E + M, M, M, M, N, N + A, R
Divisão	36	35	U, A, R, D^{28}, D + A, D + R, D + A, D + R, A, R
Raiz quadrada	112	111	U, E, $(A+R)^{108}$, A, R
Negação	2	1	U, S
Valor absoluto	2	1	U, S
Comparação de PF	3	2	U, A, R

FIGURA C.47 As latências e os intervalos de iniciação para as operações de PF dependem dos estágios da unidade de PF que determinada operação precisa utilizar.

Os valores de latência consideram que a instrução de destino é uma operação de PF; as latências são um ciclo a menos quando o destino é um store. Os estágios de pipe aparecem na ordem em que são usados para qualquer operação. A notação S + A indica um ciclo de clock em que os estágios S e A são utilizados. A notação D^{28} indica que o estágio D é usado 28 vezes em seguida.

		Ciclo de clock												
Operação	**Despacho/ stall**	**0**	**1**	**2**	**3**	**4**	**5**	**6**	**7**	**8**	**9**	**10**	**11**	**12**
Multiplicação	Despacho	U	E+M	M	M	M	N	**N+A**	**R**					
Adição	Despacho		U	S+A	A+R	R+S								
	Despacho			U	S+A	A+R	R+S							
	Despacho				U	S+A	A+R	R+S						
	Stall					U	S+A	**A+R**	**R+S**					
	Stall						U	**S+A**	**A+R**	R+S				
	Despacho							U	S+A	A+R	R+S			
	Despacho								U	S+A	A+R	R+S		

FIGURA C.48 Uma multiplicação de PF despachada no clock 0 é seguida por uma única adição de PF despachada entre os clocks 1 e 7.

A segunda coluna indica se uma instrução do tipo especificado é adiada quando é despachada n ciclos mais tarde, onde n é o número do ciclo de clock em que ocorre o estágio U da segunda instrução. O estágio ou estágios que causam um stall estão destacados. Observe que esta tabela lida apenas com a interação entre a multiplicação e uma adição despachada entre os clocks 1 e 7. Nesse caso, a adição será adiada se for despachada quatro ou cinco ciclos após a multiplicação; caso contrário, ela é despachada sem o adiamento. Observe que a adição será adiada por dois ciclos se for despachada no ciclo 4, pois no próximo ciclo de clock ela ainda entrará em conflito com a multiplicação; porém, se a adição for despachada no ciclo 5, ela será adiada por apenas um ciclo de clock, pois isso eliminará os conflitos.

		Ciclo de clock												
Operação	**Despacho/stall**	**0**	**1**	**2**	**3**	**4**	**5**	**6**	**7**	**8**	**9**	**10**	**11**	**12**
Adição	Despacho	U	S+A	A+R	R+S									
Multiplicação	Despacho		U	E+M	M	M	M	N	N+A	R				
	Despacho			U	M	M	M	M	N	N+A	R			

FIGURA C.49 Uma multiplicação despachada após uma adição sempre pode prosseguir sem stalls, pois a instrução mais curta limpa os estágios do pipeline compartilhado antes que a instrução mais longa os alcance.

		Ciclo de clock											
Operação	**Despacho/stall**	**25**	**26**	**27**	**28**	**29**	**30**	**31**	**32**	**33**	**34**	**35**	**36**
Divisão	Emitido no ciclo 0...	D	D	D	D	D	**D+A**	**D+R**	**D+A**	**D+R**	**A**	**R**	
Adição	Despacho		U	S+A	A+R	R+S							
	Despacho			U	S+A	A+R	R+S						
	Stall				U	S+A	**A+R**	**R+S**					
	Stall					U	**S+A**	**A+R**	R+S				
	Stall						U	S+A	**A+R**	**R+S**			
	Stall							U	**S+A**	**A+R**	R+S		
	Stall								U	S+A	**A+R**	**R+S**	
	Stall									U	**S+A**	**A+R**	R+S
	Despacho										U	S+A	A+R
	Despacho											U	S+A
	Despacho												U

FIGURA C.50 Uma divisão de ponto flutuante pode causar um stall para uma adição que começa perto do final da divisão.
A divisão começa no ciclo 0 e termina no ciclo 35; os 10 últimos ciclos da divisão aparecem. Como a divisão utiliza bastante o hardware de arredondamento necessário pela adição, ela adia uma adição que inicia em qualquer um dos ciclos 28-33. Observe que a adição começando no ciclo 28 será adiada para o ciclo 36. Se a adição começasse imediatamente após a divisão, ela não entraria em conflito, pois a adição poderia concluir antes que a divisão precisasse dos estágios compartilhados, exatamente como vimos na Figura C.45 para uma multiplicação e adição. Assim como na figura anterior, este exemplo assume exatamente uma adição que atinge o estágio U entre os ciclos de clock 26 e 35.

		Ciclo de clock												
Operação	**Despacho/stall**	**0**	**1**	**2**	**3**	**4**	**5**	**6**	**7**	**8**	**9**	**10**	**11**	**12**
Adição	Despacho	U	S+A	**A+R**	**R+S**									
Divisão	Stall		U	**A**	**R**	D	D	D	D	D	D	D	D	D
	Despacho			U	A	R	D	D	D	D	D	D	D	D
	Despacho				U	A	R	D	D	D	D	D	D	D

FIGURA C.51 Uma adição de precisão dupla é seguida por uma divisão de precisão dupla.
Se a divisão começar um ciclo após a adição, a divisão é adiada, mas depois disso não existe conflito.

uma multiplicação seguida por uma adição, uma adição seguida por uma multiplicação, uma divisão seguida por uma adição e uma adição seguida por uma divisão. As figuras mostram todas as posições iniciais interessantes para a segunda instrução e se essa segunda instrução será despachada ou adiada para cada posição. Naturalmente, podem existir três instruções ativas, quando as possibilidades de stalls são muito mais altas e as figuras mais complexas.

Desempenho do pipeline R4000

Nesta seção examinaremos os stalls que ocorrem para os benchmarks SPEC92 na execução da estrutura de pipeline R4000. Existem quatro causas principais de stalls ou perdas do pipeline:

1. *Stalls de load.* Os atrasos que surgem do uso de um load resultam em 1-2 ciclos após o load.
2. *Stalls de desvio condicional.* Stalls de dois ciclos em cada desvio tomado mais slots de atraso de desvio condicional não preenchidos ou cancelados.
3. *Stalls de resultado de PF.* Stalls devidos a hazards RAW para um operando de PF.
4. *Stalls estruturais de PF.* Atrasos devidos a restrições de despacho que surgem dos conflitos pelas unidades funcionais no pipeline de PF.

A Figura C.52 mostra o desmembramento do CPI para o pipeline do R4000 nos 10 benchmarks SPEC92. A Figura C.53 mostra os mesmos dados, mas em formato tabular.

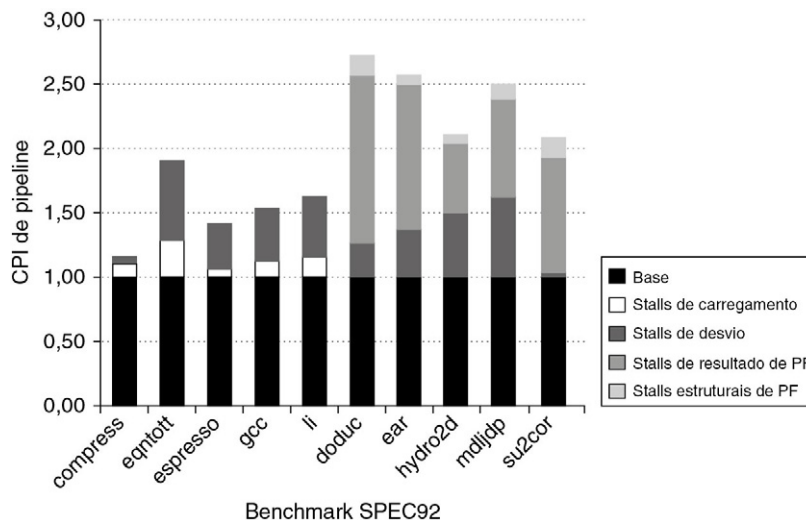



FIGURA C.52 O CPI do pipeline para 10 dos benchmarks SPEC92, assumindo uma cache perfeita. O CPI do pipeline varia de 1,2-2,8. Os cinco programas mais à esquerda são programas de inteiros, e os atrasos de desvio são um colaborador de CPI para eles. Os cinco programas mais à direita são de PF, e os stalls de resultado de PF são os principais colaboradores para eles. A Figura C.53 mostra os números usados para construir esse desenho.

A partir dos dados das Figuras C.52 e C.53, podemos ver a penalidade do pipelining mais profundo. O pipeline do R4000 possui atrasos de desvio muito mais longos do que o pipeline clássico de cinco estágios. O atraso de desvio condicional mais longo aumenta substancialmente os ciclos gastos nos desvios, especialmente para os programas de inteiros com uma frequência de desvio mais alta. Um efeito interessante para os programas de PF é que a latência das unidades funcionais de PF levam a mais stalls de resultado do que os hazards estruturais, que surgem das limitações do intervalo de iniciação e dos conflitos para as unidades funcionais a partir de diferentes instruções de PF. Assim, a redução da latência das operações de PF devem ser o primeiro alvo, em vez de mais pipelining ou replicação das unidades funcionais. Naturalmente, reduzir a latência provavelmente aumentaria os stalls estruturais, por mais que stalls estruturais em potencial sejam escondidos por trás dos hazards de dados.

Benchmark	CPI do pipeline	Stalls de load	Stalls de desvio	Stalls de resultado de PF	Stalls estruturais de PF
Compress	1,20	0,14	0,06	0,00	0,00
Eqntott	1,88	0,27	0,61	0,00	0,00
Espresso	1,42	0,07	0,35	0,00	0,00
Gcc	1,56	0,13	0,43	0,00	0,00
Li	1,64	0,18	0,46	0,00	0,00
Média de inteiros	1,54	0,16	0,38	0,00	0,00
Doduc	2,84	0,01	0,22	1,39	0,22
Mdljdp2	2,66	0,01	0,31	1,20	0,15
Ear	2,17	0,00	0,46	0,59	0,12
Hydro2d	2,53	0,00	0,62	0,75	0,17
Su2cor	2,18	0,02	0,07	0,84	0,26
Média de PF	2,48	0,01	0,33	0,95	0,18
Média geral	2,00	0,10	0,36	0,46	0,09

FIGURA C.53 **O CPI de pipeline total e as contribuições das quatro principais fontes de stalls aparecem aqui.**
As principais contribuições são os stalls de resultado de PF (tanto para desvios quanto para entradas de PF) e stalls de desvio, com loads e stalls estruturais de PF acrescentando pouco.

C.7 QUESTÕES CRUZADAS

Conjuntos de instruções RISC e eficiência do pipelining

Já discutimos as vantagens da simplicidade do conjunto de instruções na criação de pipelines. Os conjuntos de instruções simples oferecem outra vantagem. Eles facilitam o escalonamento de código para alcançar eficiência de execução em um pipeline. Para ver isso, considere um exemplo simples: suponha que precisemos somar dois valores da memória e armazenar o resultado de volta para a memória. Em alguns conjuntos de instruções sofisticados, isso exigirá uma única instrução; em outros serão necessárias duas ou três. Uma arquitetura RISC típica exigiria quatro instruções (dois loads, um add e um store). Essas instruções não podem ser escalonadas sequencialmente na maioria dos pipelines sem utilização de stalls.

Com um conjunto de instruções RISC, as operações individuais são instruções separadas e podem ser escalonadas individualmente, seja pelo compilador (usando as técnicas que discutimos anteriormente e técnicas mais poderosas, discutidas no Cap. 3), seja usando técnicas dinâmicas de escalonamento de hardware (que discutiremos em seguida e com detalhes no Cap. 3). Essas vantagens na eficiência, aliadas à maior facilidade de implementação, parecem ser tão significativas que quase todas as implementações em pipeline dos conjuntos de instruções complexas na realidade traduzem suas instruções complexas para operações simples tipo RISC, e depois escalonam e colocam em pipeline essas operações. O Capítulo 3 mostra que tanto o Pentium III quanto o Pentium 4 utilizam essa técnica.

Pipelines escalonados dinamicamente

Pipelines simples buscam uma instrução e a despacham, a menos que haja uma dependência de dados entre uma instrução já no pipeline e a instrução lida, que não pode ser ocultada com bypass ou adiantamento. A lógica de adiantamento reduz a latência efetiva do pipeline, de modo que certas dependências não resultem em hazards. Se houver um

hazard inevitável, o hardware de detecção de hazard atrasará o pipeline (começando com a instrução que usa o resultado). Nenhuma instrução nova é buscada ou despachada até que a dependência seja resolvida. Para contornar essas perdas de desempenho, o compilador pode tentar escalonar instruções para evitar o hazard; essa técnica é chamada *escalonamento de compilador* ou *escalonamento estático*.

Vários processadores mais antigos usavam outra técnica, chamada *escalonamento dinâmico*, na qual o hardware reorganiza a execução da instrução para reduzir os stalls. Esta seção oferece uma introdução mais simples ao escalonamento dinâmico, explicando a técnica de scoreboarding do CDC 6600. Alguns leitores acharão mais fácil ler este material antes de mergulhar no **[método de Tomasulo]**, mais complicado, que explicamos no Capítulo 3.

Todas as técnicas discutidas até aqui neste apêndice utilizam o despacho de instrução em ordem, o que significa que, se uma instrução for adiada no pipeline, nenhuma outra instrução poderá prosseguir. Com o despacho em ordem, se duas instruções tiverem um hazard entre elas, o pipeline será atrasado, mesmo que haja instruções mais adiante, que sejam independentes e não gerem stalls.

No pipeline MIPS desenvolvido anteriormente, hazards estruturais e de dados foram verificados durante a decodificação da instrução (ID): quando uma instrução podia ser executada corretamente, ela era despachada a partir do ID. Para permitir que uma instrução inicie a execução assim que seus operandos estejam disponíveis, mesmo que um predecessor seja adiado, temos de separar o processo de despacho em duas partes: verificar os hazards estruturais e esperar pela ausência de um hazard de dados. Decodificamos e despachamos instruções em ordem. Porém, queremos que as instruções iniciem a execução assim que seus operandos de dados estiverem disponíveis. Assim, o pipeline fará a *execução fora de ordem*, que implica *término fora de ordem*. Para implementar a execução fora de ordem, temos de dividir o estágio de pipe ID em dois estágios:

1. *Despacho.* Decodificar instruções, procurar hazards estruturais.
2. *Ler operandos.* Esperar até que não haja hazards de dados, depois ler operandos.

O estágio IF prossegue após o estágio de despacho, e o estágio EX segue o estágio de leitura de operandos, assim como no pipeline MIPS. Como no pipeline MIPS de ponto flutuante, a execução pode exigir vários ciclos, dependendo da operação. Assim, podemos ter de distinguir quando uma instrução *inicia a execução* e quando ela *termina a execução*; entre os dois tempos, a instrução está *em execução*. Isso permite que várias instruções estejam em execução ao mesmo tempo. Além dessas mudanças na estrutura do pipeline, também mudaremos o projeto da unidade funcional variando o número de unidades, a latência das operações e o pipelining da unidade funcional, a fim de explorar melhor essas técnicas de pipelining mais avançadas.

Escalonamento dinâmico com um Scoreboard

Em um pipeline escalonado dinamicamente, todas as instruções passam pelo estágio de despacho em ordem (*in-order issue*); porém, elas podem ser adiadas ou puladas no segundo estágio (ler operandos) e, portanto, entrar na execução fora de ordem. O *scoreboarding* é uma técnica para permitir que as instruções sejam executadas fora de ordem quando houver recursos suficientes e nenhuma dependência de dados; ele foi assim denominado após o scoreboard do CDC 6600, que desenvolveu essa capacidade.

Antes de vermos como o scoreboarding poderia ser usado no pipeline MIPS, é importante observar que os hazards WAR, que não existiam nos pipelines de ponto flutuante ou

inteiros do MIPS, podem surgir quando as instruções forem executadas fora de ordem. Por exemplo, considere estas sequências de código:

```
DIV.D    F0,F2,F4
ADD.D    F10,F0,F8
SUB.D    F8,F8,F14
```

Existe uma antidependência entre o ADD.D e o SUB.D: se o pipeline executar o SUB.D antes do ADD.D, ela violará a antidependência, gerando uma execução incorreta. De modo semelhante, para evitar violar as dependências de saída, os hazards WAW (p. ex., conforme ocorreria se o destino do SUB.D fosse F10) também precisam ser detectados. Conforme veremos, esses dois hazards são evitados em um scoreboard adiando-se a instrução posterior envolvida na antidependência.

O objetivo de um scoreboard é manter uma taxa de execução de uma instrução por ciclo de clock (quando não existem hazards estruturais) executando uma instrução o mais cedo possível. Assim, quando a próxima instrução a executar for adiada, outras instruções poderão ser despachadas e executadas se não dependerem de qualquer instrução ativa ou adiada. O scoreboard assume total responsabilidade pelo despacho e execução da instrução, incluindo toda detecção de hazard. Para tirar proveito da execução fora de ordem, é preciso que múltiplas instruções estejam em seu estágio EX simultaneamente. Isso pode ser conseguido com múltiplas unidades funcionais, com unidades funcionais em pipeline ou com ambos. Como essas duas capacidades — unidades funcionais em pipeline e múltiplas unidades funcionais — são basicamente equivalentes para fins de controle de pipeline, assumiremos que o processador possui múltiplas unidades funcionais.

O CDC 6600 possui 16 unidades funcionais separadas, incluindo quatro unidades de ponto flutuante, cinco unidades para referências de memória e sete unidades para operações com inteiros. Em um processador para a arquitetura MIPS, os scoreboards fazem sentido principalmente na unidade de ponto flutuante, pois a latência das outras unidades funcionais é muito pequena. Vamos supor que existam dois multiplicadores, um somador, uma unidade de divisão e uma única unidade de inteiros para todas as referências de memória, desvios e operações com inteiros. Embora esse exemplo seja mais simples do que o CDC 6600, ele é suficientemente poderoso para demonstrar os princípios sem que se tenha uma quantidade de detalhes ou que se precise de exemplos muito longos. Como o MIPS e o CDC 6600 são arquiteturas do tipo load-store, as técnicas são quase idênticas para os dois processadores. A Figura C.54 mostra como é o processador.

Cada instrução passa pelo scoreboard, onde é construído um registro das dependências de dados; essa etapa corresponde ao despacho da instrução e substitui parte da etapa ID no pipeline MIPS. O scoreboard, então, determina quando uma instrução pode ler seus operandos e iniciar a execução. Se o scoreboard decidir que a instrução não pode ser executada imediatamente, ele monitora cada mudança no hardware e decide quando a instrução *pode* ser executada. O scoreboard também controla quando uma instrução pode escrever seu resultado no registrador de destino. Assim, toda detecção e solução de hazard é centralizada no scoreboard. Veremos uma figura do scoreboard mais adiante (Fig. C.55, na página C-66), mas primeiro temos de entender as etapas no segmento de despacho e execução do pipeline.

Cada instrução passa por quatro etapas na execução. (Como estamos nos concentrando nas operações de PF, não vamos considerar uma etapa para acesso à memória.) Primeiro, vamos examinar as etapas informalmente, para depois examinar com detalhes como o scoreboard mantém as informações necessárias, que determinam quando prosseguir de

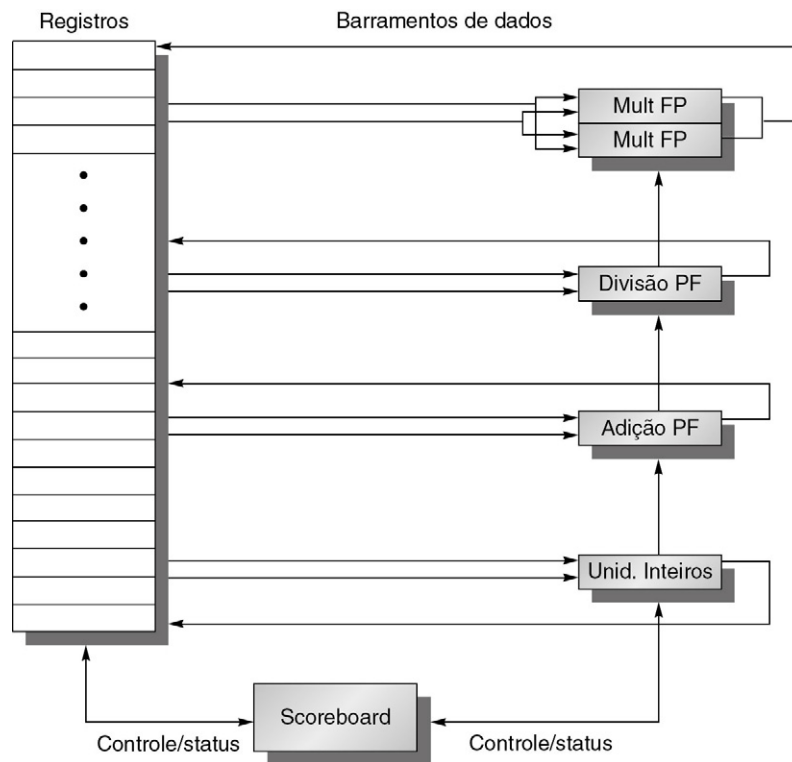



FIGURA C.54 **Estrutura básica de um processador MIPS com um scoreboard.**
A função do scoreboard é controlar a execução da instrução (linhas de controle verticais). Todos os dados fluem entre o banco de registradores e as unidades funcionais pelos barramentos (as linhas horizontais, chamadas de trunks no CDC 6600). Existem dois multiplicadores de PF, um divisor de PF e uma unidade de inteiros. Um conjunto de barramentos (duas entradas e uma saída) serve a um grupo de unidades funcionais. Os detalhes do scoreboard aparecem nas Figuras C.55 a C.58.

uma etapa para a seguinte. As quatro etapas, que substituem as etapas ID, EX e WB no pipeline MIPS padrão, são as seguintes:

1. *Despacho.* Se uma unidade funcional para a instrução estiver livre e nenhuma outra instrução ativa tiver o mesmo registrador de destino, o scoreboard despacha a instrução para a unidade funcional e atualiza sua estrutura de dados interna. Essa etapa substitui uma parte da etapa de ID no pipeline MIPS. Garantindo que nenhuma outra unidade funcional ativa deseja escrever seu resultado no registrador de destino, garantimos que os hazards WAW não podem estar presentes. Se existe um hazard estrutural ou WAW, a instrução emite stalls, e nenhuma outra instrução será despachada até que esses hazards sejam resolvidos. Quando o estágio de despacho é adiado, ele faz com que o buffer entre a busca e o despacho da instrução seja preenchido; se o buffer tiver uma única entrada, a busca de instrução é adiada imediatamente. Se o buffer é uma fila com múltiplas instruções, ele é adiado quando a fila é preenchida.
2. *Leitura de operandos.* O scoreboard monitora a disponibilidade dos operandos-fonte. Um operando-fonte está disponível se nenhuma instrução ativa despachada anteriormente tiver que atualizá-lo. Quando os operandos-fonte estão disponíveis, o scoreboard diz à unidade funcional para prosseguir e ler os operandos dos registradores e iniciar a execução. O scoreboard resolve os hazards

Instrução		Estado da instrução			
		Despacho	Leitura dos operandos	Execução completa	Escrita do resultado
L.D	F6,34(R2)	√	√	√	√
L.D	F2,45(R3)	√	√	√	
MUL.D	F0,F2,F4	√			
SUB.D	F8,F6,F2	√			
DIV.D	F10,F0,F6	√			
ADD.D	F6,F8,F2				

Nome	Estado da unidade funcional								
	Ocupado	Op	Fi	Fj	Fk	Qj	Qk	Rj	Rk
Integer	Sim	Load	F2	R3				Não	
Mult1	Sim	Mult	F0	F2	F4	Integer		Não	Sim
Mult2	Não								
Add	Sim	Sub	F8	F6	F2		Integer	Sim	Não
Divide	Sim	Div	F10	F0	F6	Mult1		Não	Sim

	Estado de resultado de registrador							
	F0	F2	F4	F6	F8	F10	F12	...	F30
FU	Mult1	Integer			Add	Divide			

FIGURA C.55 Componentes do scoreboard.

Cada instrução que foi despachada ou que está com o despacho pendente possui uma entrada na tabela de estado de instrução. Há uma entrada na tabela de estado da unidade funcional para cada unidade funcional. Quando uma instrução é despachada, o registro de seus operandos é mantido na tabela de estado da unidade funcional. Finalmente, a tabela de resultado do registrador indica qual unidade produzirá cada resultado pendente; o número de entradas é igual ao número de registradores. A tabela de estado de instrução diz que: 1) o primeiro L.D completou e escreveu seu resultado; e 2) o segundo L.D completou a execução, mas ainda não escreveu seu resultado. MUL.D, SUB.D e DIV.D foram despachados mas com stalls, esperando por seus operandos. O estado da unidade funcional diz que a primeira unidade de multiplicação está esperando pela unidade de inteiros, a unidade de adição está esperando pela unidade de inteiros e a unidade de divisão está esperando pela primeira unidade de multiplicação. A instrução ADD.D é adiada devido a um hazard estrutural; ele será resolvido quando o SUB.D concluir. Se uma entrada em uma dessas tabelas de scoreboard não estiver sendo usada, ela fica em branco. Por exemplo, o campo Rk não é usado em um load e a unidade Mult2 não é usada, pois seus campos não têm significado. Além disso, quando um operando tiver sido lido, os campos Rj e Rk são definidos como Não. A Figura C.58 mostra por que esta última etapa é crucial.

RAW dinamicamente nessa etapa, e as instruções podem ser enviadas para execução fora de ordem. Esa etapa, junto com o despacho, completa a função da etapa ID no pipeline MIPS simples.

3. *Execução.* A unidade funcional inicia a execução ao receber operandos. Quando o resultado está pronto, ela notifica o scoreboard e o avisa que completou a execução. Essa etapa substitui a etapa EX no pipeline do MIPS e utiliza múltiplos ciclos no pipeline de PF do MIPS.
4. *Escrita do resultado.* Quando o scoreboard está ciente de que a unidade funcional completou a execução, o scoreboard verifica os hazards WAR e adia a instrução terminando, se for necessário.

Existirá um hazard WAR se houver uma sequência de código como nosso exemplo anterior, com ADD.D e SUB.D, que utilizam F8. Nesse exemplo, tínhamos o código

```
DIV.D       F0,F2,F4
ADD.D       F10,F0,F8
SUB.D       F8,F8,F14
```

O ADD.D possui um operando-fonte F8, que é o mesmo registrador do destino de SUB.D. Mas ADD.D na realidade depende de uma instrução anterior. O scoreboard ainda adiará o SUB.D em seu estágio Write Result até que ADD.D leia seus operandos. Em geral, então, uma instrução terminando não pode ter permissão para escrever seus resultados quando:

- Houver uma instrução que seus operandos não tenham lido que precede (ou seja, na ordem de despacho) a instrução que está sendo completada e
- Um dos operandos é o mesmo registrador do resultado da instrução que está sendo completada.

Se esse hazard WAR não existir ou quando ele for resolvido, o scoreboard dirá à unidade funcional para armazenar seu resultado no registrador de destino. Essa etapa substitui a etapa WB no pipeline MIPS simples.

À primeira vista, pode parecer que o scoreboard terá dificuldade para separar os hazards RAW e WAR.

Como os operandos para uma instrução são lidos apenas quando os dois operandos estão disponíveis no banco de registradores, esse scoreboard não tira proveito do adiantamento. Em vez disso, os registradores só são lidos quando ambos estão disponíveis. Essa não é uma penalidade tão grande quanto você poderia pensar inicialmente. Diferentemente do nosso pipeline anterior simples, as instruções escreverão seu resultado no banco de registradores assim que completarem a execução (supondo que não haja hazards WAR), em vez de esperar por um slot de escrita atribuído estaticamente, que pode estar a vários ciclos de distância. O efeito é uma latência de pipeline reduzida e benefícios de adiantamento. Ainda existe um ciclo de latência adicional, pois os estágios de escrita de resultados e leitura de operandos não podem se sobrepor. Precisaríamos de buffers adicionais para eliminar esse overhead.

Com base em sua própria estrutura de dados, o scoreboard controla o progresso da instrução de uma etapa para a seguinte, comunicando-se com as unidades funcionais. Porém, há uma pequena complicação. Há apenas um número limitado de barramentos do operando-fonte e barramentos de resultado para o banco de registradores, o que representa um hazard estrutural. O scoreboard precisa garantir que o número de unidades funcionais permitidas para prosseguir para as etapas 2 e 4 não exceda o número de barramentos disponíveis. Não entraremos em mais detalhes sobre isso, além de mencionar que o CDC 6600 resolveu esse problema agrupando as 16 unidades funcionais em quatro grupos e dando suporte a um conjunto de barramentos, chamados *trunks de dados*, para cada grupo. Somente uma unidade em um grupo poderia ler seus operandos ou escrever seu resultado durante um clock.

Agora vejamos a estrutura de dados detalhada mantida por um scoreboard MIPS com cinco unidades funcionais. A Figura C.55 mostra como ficariam as informações do scoreboard a meio caminho da execução desta sequência de instruções simples:

```
L.D       F6,34(R2)
L.D       F2,45(R3)
MUL.D     F0,F2,F4
SUB.D     F8,F6,F2
DIV.D     F10,F0,F6
ADD.D     F6,F8,F2
```

Existem três partes no scoreboard:

1. Estado *da instrução.* Indica em qual das quatro etapas a instrução está.
2. Estado *da unidade funcional.* Indica o estado da unidade funcional. Existem nove campos para cada unidade funcional:
   - Busy. Indica se a unidade está ocupada ou não.
   - Op. Operação a realizar na unidade (p. ex., adição ou subtração).
   - Fi. Registrador destino.
   - Fj, Fk. Números de registrador-fonte.
   - Qj, Qk. Unidades funcionais produzindo registradores-fonte Fj, Fk.
   - Rj, Rk. Flags indicando quando Fj, Fk estão prontos e ainda não lidos. *Definido como Não* após os operandos serem lidos.
3. Estado *de resultado de registrador.* Indica qual unidade funcional escreverá em cada registrador, se uma instrução ativa tiver o registrador como seu destino. Esse campo é definido como um espaço em branco sempre que não houver instruções pendentes que escreverão nesse registrador.

Agora, vejamos como a sequência de código iniciada na Figura C.55 continua a execução. Depois disso, poderemos examinar com detalhes as condições que o scoreboard usa para controlar a execução.

Exemplo Considere as seguintes latências de ciclo EX (escolhidas para ilustrar o comportamento, não sendo representativas) para as unidades funcionais de ponto flutuante: adição usa dois ciclos de clock, multiplicação usa 10 ciclos de clock e divisão usa 40 ciclos de clock. Usando o segmento de código da Figura C.55 e iniciando com o ponto indicado pelo estado da instrução na Figura C.55, mostre como ficam as tabelas de estado quando MUL.D e DIV.D estão prontos para prosseguir para o estado Write Result.

Resposta Existem hazards de dados RAW do segundo L.D para MUL.D, ADD.D e SUB.D, de MUL.D para DIV.D e de SUB.D para ADD.D. Existe um hazard de dados WAR entre DIV.D e ADD.D e SUB.D. Finalmente, existe um hazard estrutural na unidade funcional de adição para ADD.D e SUB.D. O formato das tabelas quando MUL.D e DIV.D estão prontos para escrever seus resultados aparece nas Figuras C.56 e C.57, respectivamente.

Agora, podemos ver com detalhes como funciona o scoreboard, examinando o que precisa acontecer para o scoreboard permitir que cada instrução prossiga. A Figura C.58 mostra o que o scoreboard exige para cada instrução avançar e a ação de manutenção necessária quando a instrução avança. O scoreboard registra informações de especificador de operando, como os números dos registradores. Por exemplo, temos de registrar os registradores-fonte quando uma instrução é despachada. Como nos referimos ao conteúdo de um registrador como Regs[D], onde D é um nome de registrador, não existe ambiguidade. Por exemplo, Fj[FU] S1] faz com que o *nome* de registrador S1 seja colocado em Fj[FU], em vez do *conteúdo* do registrador S1.

Instrução		Estado da instrução			
		Despacho	Ler operandos	Execução completa	Resultado da escrita
L.D	F6,34(R2)	√	√	√	√
L.D	F2,45(R3)	√	√	√	√
MUL.D	F0,F2,F4	√	√	√	
SUB.D	F8,F6,F2	√	√	√	√
DIV.D	F10,F0,F6	√			
ADD.D	F6,F8,F2	√	√	√	

	Estado da unidade funcional								
Nome	Ocupado	Op	Fi	Fj	Fk	Qj	Qk	Rj	Rk
Integer	Não								
Mult1	Sim	Mult	F0	F2	F4			Não	Não
Mult2	Não								
Add	Sim	Add	F6	F8	F2			Não	Não
Divide	Sim	Div	F10	F0	F6	Mult1		Não	Sim

	Estado de resultado de registrador							
	F0	F2	F4	F6	F8	F10	F12	...	F30
FU	Mult1			Add		Divide			

FIGURA C.56 **Tabelas do scoreboard imediatamente antes que o MUL.D vá para a escrita do resultado.**
O DIV.D ainda não leu qualquer um de seus operandos, pois tem uma dependência no resultado da multiplicação. O ADD.D leu seus operandos e está em execução, embora fosse forçado a esperar até que o SUB.D terminasse, para obter a unidade funcional. ADD.D não pode prosseguir para escrever o resultado, devido ao hazard WAR em F6, que é usado pelo DIV.D. Os campos Q são relevantes apenas quando uma unidade funcional está esperando por outra unidade.

Os custos e os benefícios do scoreboarding são considerações interessantes. Os projetistas do CDC 6600 mediram uma melhoria de desempenho de 1,7 para programas em FORTRAN e 2,5 para a linguagem assembly codificada à mão. Porém, isso foi medido nos dias anteriores ao escalonamento do pipeline de software, memória principal semicondutora e caches (que reduzem o tempo de acesso à memória). O scoreboard no CDC 6600 tinha tanta lógica quanto uma das unidades funcionais, que é surpreendentemente baixa. O custo principal estava no grande número de barramentos — cerca de quatro vezes mais do que seria necessário se a CPU só executasse instruções em ordem (ou se só iniciasse uma instrução por ciclo de execução). O interesse recente cada vez maior no escalonamento dinâmico é motivado pelas tentativas de despachar mais instruções por clock (de modo que o custo de mais barramentos precisa ser pago de qualquer forma) e por ideias como especulação (explorada na Seção 4.7), que se baseavam naturalmente no escalonamento dinâmico.

Instrução		Estado da instrução			
		Despacho	Ler operandos	Execução completa	Resultado da escrita
L.D	F6,34(R2)d	√	√	√	√
L.D	F2,45(R3)	√	√	√	√
MUL.D	F0,F2,F4	√	√	√	√
SUB.D	F8,F6,F2	√	√	√	√
DIV.D	F10,F0,F6	√	√	√	
ADD.D	F6,F8,F2	√	√	√	√

	Estado da unidade funcional								
Nome	Ocupado	Op	Fi	Fj	Fk	Qj	Qk	Rj	Rk
Integer	Não								
Mult1	Sim	Mult	F0	F2	F4			Não	Não
Mult2	Não								
Add	Sim	Add	F6	F8	F2			Não	Não
Divide	Sim	Div	F10	F0	F6			Não	Sim

	Estado de resultado de registrador							
	F0	F2	F4	F6	F8	F10	F12	...	F30
FU	Mult 1			Add		Divide			

FIGURA C.57 **Tabelas do scoreboard imediatamente antes que o `DIV.D` vá para a escrita de resultados.**
O `ADD.D` foi capaz de concluir assim que `DIV.D` passou pelos operandos de leitura e obteve uma cópia de F6. Só resta o `DIV.D` para terminar.

Estado da instrução	Espera até	Manutenção
Despacho	Não ocupado [FU] e não Result [D]	`Busy[FU]← yes; Op[FU] ←op; Fi[FU] ←D; Fj[FU] ←S1; Fk[FU] ←S2; Qj←Result[S1]; Qk←Result[S2]; Rj←not Qj; Rk← not Qk; Result[D] ←FU;`
Ler operandos	Rj e Rk	`Rj←No; Rk←No; Qj←0; Qk←0`
Execução completa	Unidade funcional pronta	
Escrita de resultados	∀*f*((Fj[*f*] ≠ Fi[FU] ou Rj[*f*] = No) & (Fk[*f*] ≠ Fi[FU] ou Rk[*f*] = No))	`∀f(if Qj[f]=FU then Rj[f] ←Yes); ∀f(if Qk[f]=FU then Rk[f] ←Yes); Result[Fi[FU]] ←0; Busy[FU] ←No`

FIGURA C.58 **Verificações e ações de manutenção exigidas para cada etapa na execução da instrução.**
FU significa a unidade funcional utilizada pela instrução, D é o nome do registrador de destino, S1 e S2 são os nomes dos registradores-fonte e op é a operação a ser feita. Para acessar a entrada do scoreboard chamada Fj para a unidade funcional FU, usamos a notação Fj[FU]. Result[D] é o nome da unidade funcional que escreverá no registrador D. O teste no caso de resultado de escrita impede a escrita quando existe um hazard WAR, que existirá se outra instrução tiver o destino dessa instrução (Fi[FU]) como fonte (Fj[f] ou Fk[f]) e se alguma outra instrução tiver escrito no registrador ((Rj = Sim ou Rk = Sim). A variável f é usada para qualquer unidade funcional.

Um scoreboard utiliza o ILP disponível para reduzir o número de stalls que surgem das dependências de dados verdadeiras do programa. Eliminando stalls, um scoreboard é limitado por diversos fatores:

1. *Quantidade de paralelismo disponível entre as instruções.* Isso determina se instruções independentes podem ser encontradas para executar. Se cada instrução depende da sua predecessora, nenhum esquema de endereçamento dinâmico pode reduzir os stalls. Se as instruções no pipeline simultaneamente tiverem de ser escolhidas a partir do mesmo bloco básico (como acontecia no 6600), esse limite provavelmente será muito rigoroso.
2. *Número de entradas no scoreboard.* Isso determina até que ponto adiante o pipeline pode procurar instruções independentes. O conjunto de instruções examinadas como candidatas para execução em potencial é chamado *janela*. O tamanho do scoreboard determina o tamanho da janela. Nesta seção, consideramos que uma janela não se estende além de um desvio, de modo que a janela (e o scoreboard) sempre contém código linear de um único bloco básico. O Capítulo 2 mostra como a janela pode ser estendida além de um desvio.
3. *Número e tipos de unidades funcionais.* Isso determina a importância dos hazards estruturais, que podem aumentar quando o escalonamento dinâmico for usado.
4. *Presença de antidependências e dependências de saída.* Estes levam a stalls WAR e WAW.

O Capítulo 3 focaliza as técnicas que atacam o problema de expor e utilizar melhor o ILP disponível. O segundo e terceiro fatores podem ser atacados pelo aumento no tamanho do scoreboard e pelo número de unidades funcionais; porém, essas mudanças têm implicações no custo e também podem afetar o tempo do ciclo. Os hazards WAW e WAR se tornam mais importantes nos processadores escalonados dinamicamente, pois o pipeline expõe mais dependências de nome. Os hazards WAW também se tornam mais importantes se usarmos o escalonamento dinâmico com um esquema de previsão de desvio que permita a sobreposição de múltiplas iterações de um loop.

C.8 FALÁCIAS E ARMADILHAS

Armadilha. *Sequências de execução inesperadas podem causar hazards inesperados.*

À primeira vista, hazards WAW parece que nunca deveriam ocorrer em uma sequência de código, pois nenhum compilador sequer geraria duas escritas no mesmo registrador sem uma leitura intercalada. Mas eles podem ocorrer quando a sequência é inesperada. Por exemplo, a primeira escrita poderia estar no slot de atraso de um desvio condicional tomado quando o escalonador considerou que o desvio não seria tomado. Aqui está uma sequência de código que causaria isso:

```
          BNEZ      R1,foo
          DIV.D     F0,F2,F4; movido para slot de atraso
                    ;da sequência
          .....
          .....
foo:      L.D       F0,qrs
```

Se o desvio for tomado, antes que o `DIV.D` possa ser concluído o `L.D` alcançará WB, causando um hazard WAW. O hardware precisa detectar isso e pode adiar o despacho do `L.D`. Outra maneira como isso pode acontecer é se a segunda escrita

estiver em uma rotina de trap. Isso ocorre quando uma instrução que intercepta e está escrevendo resultados continua e termina após uma instrução que escreve no mesmo registrador na rotina de tratamento de trap. O hardware também precisa detectar e impedir isso.

Armadilha. *Pipelining muito grande pode ter impacto em outros aspectos de um projeto, levando a um custo-desempenho geral pior.*

O melhor exemplo desse fenômeno vem de duas implementações do VAX, o 8600 e o 8700. Quando o 8600 foi entregue inicialmente, ele tinha um tempo de ciclo de 80 ns. Subsequentemente, uma versão reprojetada, chamada 8650, com um clock de 55 ns, foi introduzida. O 8700 tem um pipeline muito mais simples, que opera no nível de microinstrução, gerando uma CPU menor, com um ciclo de clock mais rápido, de 45 ns. O resultado geral é que o 8650 possui uma vantagem de CPI de cerca de 20%, mas o 8700 tem uma taxa de clock que é cerca de 20% mais rápida. Assim, o 8700 alcança o mesmo desempenho com muito menos hardware.

Armadilha. *Avaliando o escalonamento dinâmico ou estático com base no código não otimizado.*

O código não otimizado — contendo loads, stores e outras operações redundantes, que poderiam ser eliminadas por um otimizador — é muito mais fácil de escalonar do que o código otimizado "rígido". Isso acontece para o escalonamento de atrasos de controle (com desvios adiados) e atrasos que surgem de hazards RAW. No gcc rodando em um R3000, que tem um pipeline quase idêntico à da Seção C.1, a frequência dos ciclos de clock ociosos aumenta em 18% do código não otimizado e escalonado para o código otimizado e escalonado. Naturalmente, o programa otimizado é muito mais rápido, pois tem menos instruções. Para avaliar de forma justa um escalonador em tempo de compilação ou o escalonamento dinâmico em tempo de execução, você precisa usar o código otimizado, pois no sistema real terá um bom desempenho de outras otimizações, além do escalonamento.

C.9 COMENTÁRIOS FINAIS

No início da década de 1980, o pipelining era uma técnica reservada principalmente para supercomputadores e grandes mainframes de milhões de dólares. Em meados da mesma década, os primeiros microprocessadores em pipeline apareceram e ajudaram a transformar o mundo da computação, permitindo que os microprocessadores ultrapassem os minicomputadores em desempenho e, por fim, assumam o lugar e sejam superiores aos mainframes. No início da década de 1990, os microprocessadores embarcados de alto nível adotaram o pipelining, e os desktops se voltaram para o uso das técnicas sofisticadas de despacho múltiplo, escalonadas dinamicamente, discutidas no Capítulo 3. O material neste apêndice, que foi considerado razoavelmente avançado para alunos formados quando este texto apareceu inicialmente em 1990, agora é considerado material básico de curso de formação e pode ser encontrado em processadores custando menos de US$ 10!

C.10 PERSPECTIVAS HISTÓRICAS E REFERÊNCIAS

A Seção L.5 (disponível on-line) contém uma discussão sobre o desenvolvimento do pipelining e paralelismo em nível de instrução. Oferecemos diversas referências para leitura adicional e uma exploração desses tópicos.

EXERCÍCIOS ATUALIZADOS POR DIANA FRANKLIN

C.1 [15/15/15/15/25/10/15] <C.2> Use o seguinte fragmento de código:

```
Loop:    LD     R1,0(R2)    ;Carrega R1 do endereço 0+R2
         DADDI  R1,R1,#1    ;R1=R1+1
         SD     R1,0,(R2)   ;armazena R1 no endereço 0+R2
         DADDI  R2,R2,#4    ;R2=R2+4
         DSUB   R4,R3,R2    ;R4=R3-R2
         BNEZ   R4,Loop     ;desvio para o Loop se R4!=0
```

Suponha que o valor inicial de R3 seja R2 + 396.

a. [15] <C.2> Hazards de dados são causados por dependências de dados no código. Se uma dependência que causa um hazard depende da implementação da máquina (p. ex., número de estágios no pipeline). Liste todas as dependências de dados no código anterior. Anote o registrador, instrução-fonte e instrução de destino. Por exemplo, há uma dependência de dados para o registrador R1 vinda do `LD` para o `DADDI`.

b. [15] <C.2> Mostre a temporização dessa sequência de instruções para o pipeline RISC de cinco estágios sem nenhum hardware de adiantamento ou bypassing, mas supondo que uma leitura e uma escrita de registrador no mesmo ciclo de clock "adiantam" através do banco de registradores, como mostrado na Figura C.6. Use uma tabela de temporização do pipeline como o da Figura C.5. Suponha que o desvio seja tratado esvaziando o pipeline. Se todas as referências de memória levam um ciclo, de quantos ciclos esse loop precisa para ser executado?

c. [15] <C.2> Mostre a temporização dessa sequência de instruções para o pipeline RISC de cinco estágios com hardware completo de adiantamento e bypassing. Use uma tabela de temporização de pipeline como mostrado na Figura C.5. Suponha que o desvio seja tratado prevendo-o como não sendo tomado. Se todas as referências de memória levam um ciclo, quantos ciclos esse loop precisa para ser executado?

d. [15] <C.2> Mostre a temporização dessa sequência de instruções para o pipeline RISC de cinco estágios com hardware completo de adiantamento e bypassing. Use uma tabela de temporização de pipeline como mostrado na Figura C.5. Suponha que o desvio seja tratado prevendo-o como sendo tomado. Se todas as referências de memória levam um ciclo, de quantos ciclos esse loop precisa para ser executado?

e. [25] <C.2> Processadores de alto desempenho têm pipelines muito profundos — mais de 15 estágios. Imagine que você tenha um pipeline de 10 estágios no qual cada estágio de um pipeline de cinco estágios foi dividido em dois. A pegadinha é que, para o adiantamento de dados, os dados são adiantados do final de um par de estágios para o começo dos dois estágios onde eles são necessários. Por exemplo, dados são adiantados da saída do segundo estágio de execução para a entrada do primeiro estágio de execução, ainda causando um atraso de um ciclo. Mostre a temporização dessa sequência de instruções para o pipeline RISC de 10 estágios com hardware completo de adiantamento e bypassing. Use uma tabela de temporização de pipeline como mostrado na Figura C.5. Suponha que o desvio seja tratado prevendo-o como sendo tomado.

Se todas as referências de memória levam um ciclo, de quantos ciclos esse loop precisa para ser executado?

f. [10] <C.2> Suponha que no pipeline de cinco estágios, o estágio mais longo requer 0,8 ns e o atraso do registrador de pipeline é de 0,1 ns. Qual é o tempo do ciclo de clock do pipeline de cinco estágios? Se o pipeline de 10 estágios tiver todos os estágios divididos pela metade, qual é o tempo do ciclo para a máquina de 10 estágios?

g. [15] <C.2> Usando suas respostas dos itens *d* e *e*, determine os ciclos por instrução (CPI) para o loop em um pipeline de cinco estágios e um pipeline de 10 estágios. Tenha certeza de contar somente a partir de quando a primeira instrução atinge o estágio write-back até o final. Não conte o startup da primeira instrução. Usando o tempo de ciclo de clock calculado do item *f*, calcule o tempo médio de execução de instrução para cada máquina.

C.2 [15/15] <C.2> Suponha as seguintes frequências de desvio (como porcentagens de todas as instruções):

Desvios condicionais	15%
Saltos e chamadas	1%
Desvios condicionais tomados	60% são tomados

a. [15] <C.2> Estamos examinando um pipeline com profundidade de quatro, onde o desvio é resolvido no fim do segundo ciclo para desvios incondicionais e no fim do terceiro ciclo para desvios condicionais. Supondo que somente o primeiro estágio de pipe possa sempre ser realizado independentemente de para onde o desvio vai e ignorando outros stalls de pipeline, quão mais rápida a máquina seria sem nenhum hazard de desvio?

b. [15] <C.2> Agora suponha um processador de alto desempenho no qual tenhamos um pipeline com profundidade de 15, onde o desvio é resolvido no fim do quinto ciclo para desvios incondicionais e no fim do décimo ciclo para desvios condicionais. Supondo que somente o primeiro estágio de pipe possa sempre ser realizado independentemente de para onde o desvio vai e ignorando outros stalls de pipeline, quão mais rápida a máquina seria sem nenhum hazard de desvio?

C.3 [5/15/10/10] <C.2> Nós começamos com um computador com uma implementação de ciclo único. Quando os estágios são divididos por funcionalidade, não requerem exatamente a mesma quantidade de tempo. A máquina original tinha um tempo de ciclo de clock de 7 ns. Depois que os estágios foram divididos, os tempos medidos foram IF, 1 ns; ID, 1,5 ns; EX, 1 ns; MEM, 2 ns; e WB, 1,5 ns. O atraso do registrador de pipeline é de 0,1 ns.

a. [5] <C.2> Qual é o tempo de ciclo de clock da máquina com pipeline em 54 estágios?

b. [15] <C.2> Se houver um atraso a cada quatro instruções, qual será o CPI da nova máquina?

c. [10] <C.2> Qual é o ganho de velocidade da máquina com pipeline em relação à máquina de ciclo único?

d. [10] <C.2> Se a máquina com pipeline tivesse um número infinito de estágios, qual seria seu ganho de velocidade em relação à máquina de ciclo único?

C.4 [15] <C.1, C.2> Uma implementação reduzida de hardware do clássico pipeline de cinco estágios pode usar o hardware do estágio EX para realizar uma comparação de instrução de desvio e não entregar realmente o PC-alvo do desvio para o estágio IF até o ciclo de clock no qual a instrução de desvio atinge o estágio MEM. Stalls de hazard de controle podem ser reduzidos resolvendo as instruções de desvio no ID, mas melhorar o desempenho em um aspecto pode reduzir o desempenho em outras circunstâncias. Escreva um pequeno trecho de código no qual calcular o desvio no estágio ID causa um hazard de dados, mesmo com avanço de dados.

C.5 [12/13/20/20/15/15] <C.2, C.3> Para esses problemas, vamos explorar um pipeline para uma arquitetura registrador-memória. A arquitetura tem dois formatos de instrução: um registrador-registrador e um registrador-memória. Há um modo de endereçamento de memória única (offset + registrador base). Há um conjunto de operações de ALU com o formato:

```
ALUop Rdest, Rsrc1, Rsrc2
```

ou

```
ALUop Rdest, Rsrc1, MEM
```

onde o `ALUop` é um dos seguintes: add, subtract, AND, OR, load (Rsrc1 ignored) ou store. Rsrc ou Rdest são registradores. MEM é um par de registradores de base e de offset. Desvios condicionais usam uma comparação completa de dois registradores e são relacionados ao PC. Suponha que essa máquina possua um pipeline de modo que uma nova instrução seja iniciada a cada ciclo de clock. A estrutura de pipeline, similar à usada no micropipeline VAX 8700 (Clark, 1987), é

IF	**RF**	**ALU1**	**MEM**	**WB**						
	IF	RF	ALU1	MEM	ALU2	WB				
		IF	RF	ALU1	MEM	ALU2	WB			
			IF	RF	ALU1	MEM	ALU2	WB		
				IF	RF	ALU1	MEM	ALU2	WB	
					IF	RF	ALU1	MEM	ALU2	WB

O primeiro estágio da ALU é usado para cálculo de endereço efetivo para referências de memória e desvios. O segundo ciclo de ALU é usado para operações e comparação de desvio. O RF é um ciclo de decodificação e de busca de registrador. Suponha que, quando uma leitura de registrador e uma escrita de registrador do mesmo registrador ocorrem no mesmo clock, os dados de escrita são adiantados.

a. [12] <C.2> Encontre o número de somadores necessário, contando qualquer somador ou incrementador. Mostre uma combinação de instruções e estágios de pipe que justifiquem essa resposta. Você precisa dar só uma combinação que maximize a contagem de somador.

b. [13] <C.2> Encontre o número de portas de leitura e escrita de registrador e portas de leitura e escrita necessárias de memória. Prove que sua resposta está correta mostrando uma combinação de instruções e estágio de pipeline indicando

a instrução e número de portas de leitura e portas de escrita necessárias para essa instrução.

c. [20] <C.3> Determine quaisquer adiantamentos de dados para quaisquer ALUS que serão necessários. Suponha que existam ALUs separadas para os estágios de pipe ALU1 e ALU2. Coloque todos os avanços entre ALUs necessários para evitar ou reduzir stalls. Mostre o relacionamento entre as duas instruções envolvidas no avanço usando o formato da tabela na Figura C.26, mas ignorando as duas últimas colunas. Tenha certeza de considerar o avanço através de uma instrução que esteja intervindo — por exemplo,

```
ADD            R1, ...
qualquer instrução
ADD            ..., R1, ...
```

d. [20] <C.3> Mostre todos os requisitos de adiantamento de dados necessários para evitar ou reduzir stalls quando a unidade-fonte ou de destino não for uma ALU. Use o mesmo formato da Figura C.26, ignorando novamente as duas últimas colunas. Lembre-se de adiantar para e das referências de memória.

e. [15] <C.3> Mostre todos os hazards restantes que envolvem pelo menos uma unidade diferente de uma ALU como unidade-fonte ou de destino. Use uma tabela como a mostrada na Figura C.25, mas substitua a última coluna pelos comprimentos dos hazards.

f. [15] <C.2> Mostre todos os hazards de controle dando exemplos e declare a duração do stall. Use um formato como o da Figura C.11, descrevendo cada exemplo.

C.6 [12/13/13/15/15] <C.1, C.2, C.3> Vamos agora adicionar suporte a operações ALU registrador-memória ao clássico pipeline RISC de cinco estágios. Para compensar esse aumento em complexidade, *todo* o endereçamento de memória será restrito a registrar indiretamente (p. ex., todos os endereços são simplesmente um valor contido em um registrador, nenhum offset ou deslocamento poderá ser adicionado ao valor do registrador). Por exemplo, a instrução registrador-memória `ADD R4, R5, (R1)` significa adicionar o conteúdo do registrador R5 ao conteúdo do local de memória com endereço igual ao valor do registrador R1 e colocar a soma no registrador R4. Operações registrador-registrador da ALU não são modificadas. Os seguintes itens se aplicam ao pipeline RISC de inteiros:

a. [12] <C.1> Liste uma ordem rearranjada dos cinco estágios tradicionais do pipeline RISC que vão suportar operações registrador-memória implementadas exclusivamente por endereçamento indireto de registrador.

b. [13] <C.2, C.3> Descreva novos caminhos de avanço que são necessários para o pipeline rearranjado, declarando a fonte, o destino e as informações transferidas em cada novo caminho necessário.

c. [13] <C.2, C.3> Para os estágios reordenados do pipeline RISC, que novos hazards de dados são criados por esse modo de endereçamento? Forneça uma sequência de instruções ilustrando cada novo hazard.

d. [15] <C.3> Liste todos os modos em que o pipeline RISC com operações de ALU registrador-memória podem ter um número de instruções para um dado programa diferente do que o pipeline RISC original. Forneça um par de

sequências de instruções específicas, um para o pipeline original e um para o pipeline rearranjado, para ilustrar cada caminho.

e. [15] <C.3> Suponha que todas as instruções levem um ciclo de clock por estágio. Liste todos os modos em que o RISC registrador-memória pode ter um CPI diferente para um dado programa em comparação ao pipeline RISC original.

C.7 [10/10] <C.3> Neste problema vamos explorar como aprofundar o pipeline afeta o desempenho de dois modos: ciclo de clock mais rápido e stalls maiores devido a hazards de dados e controle. Suponha que a máquina original seja um pipeline de cinco estágios com um ciclo de clock de 1 ns e a segunda máquina seja um pipeline de 12 estágios com um ciclo de clock de 0,6 ns. O pipeline de cinco estágios tem um stall devido a um hazard de dados a cada cinco instruções, enquanto o pipeline de 12 estágios tem três stalls a cada oito instruções. Além disso, os desvios constituem 20% da instrução, e a taxa de previsão incorreta para as duas máquinas é de 5%.

a. [10] <C.3> Qual é o ganho de velocidade do pipeline de 12 estágios em relação ao pipeline de cinco estágios, levando em conta somente os hazards de dados?

b. [10] <C.3> Se a penalidade de erro de previsão de desvio para a primeira máquina for de dois ciclos, mas para a segunda máquina for de cinco ciclos, quais serão os CPIs para cada uma, levando em conta os stalls devidos aos erros de previsão de desvio?

C.8 [15] <C.5> Crie uma tabela mostrando a lógica de avanço para o pipeline de inteiros R4000 usando o mesmo formato mostrado na Figura C.26. Inclua somente as instruções MIPS que consideramos na Figura C.26.

C.9 [15] <C.5> Crie uma tabela mostrando a detecção de hazards do pipeline de inteiros R4000 usando o mesmo formato mostrado na Figura C.25. Inclua somente as instruções MIPS que consideramos na Figura C.26.

C.10 [25] <C.25> Suponha que o MIPS tenha somente um conjunto de registradores. Construa a tabela de avanço para as instruções de PF e de inteiros usando o formato da Figura C.26. Ignore as divisões de PF e de inteiros.

C.11 [15] <C.5> Construa uma tabela como a mostrada na Figura C.25 para verificar stalls WAW no pipeline MIPS de PF da Figura C.35. Não considere divisões de PF.

C.12 [20/22/22] <C.4, C.6> Neste exercício vamos examinar como um loop comum que manipula vetor é executado em versões escalonadas estática e dinamicamente do pipeline MIPS. O loop é o DAXPY (discutido em detalhes no Apêndice G) e a operação central em eliminação gaussiana. O loop implementa a operação de vetor Y = a * X +Y para um vetor de tamanho 100. Este é o código MIPS para o loop:

```
foo:  L.D        F2, 0(R1)         ; carrega X(i)
      MUL.D      F4, F2, F0        ; multiplica a*X(i)
      L.D        F6, 0($2)         ; carrega Y(i)
      ADD.D      F6, F4, F6        ; soma a*X(i) + Y(i)
      S.D        0(R2), F6         ; armazena Y(i)
      DADDIU     R1, R1, #8        ; incrementa o índice de X
      DADDIU     R2, R2, #8        ; incrementa o índice de Y
      SGTIU      R3, R1, done      ; testa se está completo
      BEQZ       R3, foo           ; loop se não está completo
```

Para os itens *a* a *c*, suponha que as operações de inteiro sejam despachadas e completadas em um ciclo de clock (incluindo os loads) e que seus resultados sejam totalmente contornados. Ignore o atraso de desvio condicional. Você vai usar as latências de PF (somente) mostradas na Figura C.34, mas suponha que a unidade de PF seja completamente pipelined. Para os scoreboards a seguir, suponha que uma instrução aguardando por um resultado de outra unidade funcional possa passar por operandos de leitura ao mesmo tempo que o resultado é escrito. Suponha também que uma instrução na conclusão de WR permita que uma instrução ativa no momento que esteja esperando pela mesma unidade funcional despache no mesmo ciclo de clock no qual a primeira instrução completa o WR.

a. [20] <C.5> Para este problema use o pipeline MIPS da Seção C.5 com as latências de pipeline da Figura C.34, mas uma unidade de PF totalmente pipelined, de modo que o intervalo de iniciação seja de 1. Trace um diagrama de tempo, similar ao da Figura C.37, mostrando a execução de cada instrução. Quantos ciclos de clock leva cada iteração de loop, contando a partir de quando a primeira instrução entra no estágio WB até quando a última instrução entra no estágio WB?

b. [22] <C.6> Usando o código MIPS para o DAXPY acima, mostre o estado das tabelas de scoreboard (como na Figura C.56) quando a instrução SGTIU atinge a escrita do resultado. Suponha que os operandos de despacho e escrita levem um ciclo cada um. Suponha que haja uma unidade funcional de inteiros que leve somente um único ciclo de execução (a latência de uso é de 0 ciclos, incluindo loads e store). Suponha a configuração da unidade de PF da Figura C.54 com as latências de PF da Figura C.34. O desvio não deve ser incluído no scoreboard.

c. [22] <C.6> Usando o código MIPS para DAXPY `acima`, suponha um scoreboard com as unidades funcionais de PF descritas na Figura C.54, mais uma unidade funcional de inteiros (também usada para load-store). Considere as latências mostradas na Figura C.59. Mostre o estado do scoreboard (como na Fig. C.56) quando o desvio despacha pela segunda vez. Suponha que o desvio foi previsto corretamente como tomado e levou um ciclo. Quantos ciclos de clock leva cada iteração de loop? Você pode ignorar quaisquer conflitos de porta/barramento.

C.13 [25] <C.8> É fundamental que o scoreboard seja capaz de distinguir entre hazards RAW e WAW, porque um hazard WAR requer o stall da instrução realizando a escrita

Instrução produzindo o resultado	Instrução usando o resultado	Latência em ciclos de clock
Multiplicação de PF	PF ALU Op	6
Soma de PF	PF ALU Op	4
Multiplicação de PF	Store de PF	5
Soma de PF	Store de PF	3
Operação de inteiro (incluindo load)	Qualquer	0

FIGURA C.59 Latências de pipeline onde a latência é um número.

até que a instrução lendo um operando inicie a execução, mas um hazard RAW requer o atraso da instrução de leitura até que a instrução de escrita seja finalizada — o oposto. Por exemplo, considere a sequência:

```
MUL.D     F0,F6,F4
DSUB.D    F8,F0,F2
ADD.D     F2,F10,F2
```

O DSUB.D depende do MUL.D (um hazard RAW); assim, deve-se permitir que o MUL.D seja completado antes do DSUB.D. Se o MUL.D fosse atrasado para o DSUB.D devido à inabilidade de distinguir entre hazards RAW e WAW, o processador ficaria travado. Essa sequência contém um hazard WAR entre o ADD.D e o DSUB.D, e o ADD.D, e não podemos permitir que o ADD.D seja completado até que o DSUB.D comece sua execução. A dificuldade está em distinguir o hazard RAW entre MUL.D e DSUB.D, e o hazard WAR entre o DSUB.D e o ADD.D. Para ver por que o cenário de três instruções é importante, trace o tratamento de cada instrução, estágio por estágio, através de despacho, leitura de operandos, execução e escrita do resultado. Suponha que cada estágio do scoreboard diferente da execução leve um ciclo de clock. Suponha que a instrução MUL.D requeira três ciclos de clock para ser executada e que as instruções DSUB.D e ADD.D levem um ciclo cada uma para serem executadas. Por fim, suponha que o processador tenha duas unidades funcionais de multiplicação e duas unidades funcionais de adição. Apresente o traço como a seguir.

1. Crie uma tabela com os títulos de coluna Instrução, Despacho, Operandos de leitura, Execução, Escrita do Resultado e Comentário. Na primeira coluna, liste as instruções na ordem do programa (seja generoso com o espaço entre as instruções. Células maiores de tabela vão conter melhor os resultados da sua análise). Comece a tabela escrevendo um 1 na coluna de despacho da linha da instrução MUL.D para mostrar que o MUL.D completa o estágio de desempenho no ciclo de clock 1. Depois, preencha as colunas de estágio da tabela através do ciclo no qual o scoreboard faz o stall de uma função pela primeira vez.
2. Para uma instrução em stall, escreva as palavras "esperando pelo ciclo de clock X", onde X é o número do ciclo de clock atual, na coluna apropriada da tabela para mostrar que o scoreboard está resolvendo um hazard RAW ou WAW pelo stall desse estágio. Na coluna Comentário, declare o tipo de hazard e a instrução dependente que está causando a espera.
3. Adicionando as palavras "completa com o ciclo de clock Y" a uma entrada "de espera" da tabela, preencha o restante da tabela ao longo do tempo, quando todas as instruções forem completadas. Para uma instrução que sofreu stall, adicione uma descrição à coluna Comentários dizendo por que a espera acabou e como o travamento foi evitado. (*Dica*: Pense em como os hazards WAE são impedidos e o que isso implica em relação às sequências de instrução ativas). Observe a ordem de conclusão das três instruções em relação à sua ordem de programa.

C.14 [10/10/10] <C.5> Para este problema, você vai criar uma série de pequenos trechos de código que ilustram os problemas que surgem quando se usam diferentes

unidades funcionais com diferentes latências. Para cada uma, trace um diagrama de tempo similar ao da Figura C.38 que ilustre cada conceito e indique claramente o problema.

a. [10] <C.5> Demonstre, usando código diferente do usado na Figura C.38, o hazard estrutural de ter o hardware para somente um estágio MEM e WB.

b. [10] <C.5> Demonstre um hazard WAW requerendo um stall.

Referências

Adve, S. V., e K. Gharachorloo [1996]. "Shared memory consistency models: A tutorial", *IEEE Computer* 29:12 (Dezembro), 66-76.

Adve, S. V., e M. D. Hill [1990]. "Weak ordering—a new definition", *Proc. 17th Annual Int'l. Symposium on Computer Architecture (ISCA)*,Maio 28-31, 1990, Seattle, Wash., 2-14.

Agarwal, A. [1987]. "Analysis of Cache Performance for Operating Systems and Multi-programming", Tese de Doutorado, Tech. Rep. No. CSL-TR-87-332, Stanford University, Palo Alto, Calif.

Agarwal, A. [1991]. "Limits on interconnection network performance", *IEEE Trans. on Parallel and Distributed Systems* 2:4 (Abril), 398-412.

Agarwal, A., e S. D. Pudar [1993]. "Column-associative caches: A technique for reduc-ing the miss rate of direct-mapped caches", *20th Annual Int'l. Symposium on Com-puter Architecture (ISCA)*, 16-19 de Maio de 1993, San Diego, Calif. Também aparece em *Computer Architecture News* 212 (Maio), 179-190, 1993.

Agarwal, A., R. Bianchini, D. Chaiken, K. Johnson, e D. Kranz [1995]. "The MIT Alewife machine: Architecture and performance", *Int'l. Symposium on Computer Architecture* (Denver, Colo.), Junho, 2-13.

Agarwal, A., J. L. Hennessy, R. Simoni, e M. A. Horowitz [1988]. "An evaluation of directory schemes for cache coherence", *Proc. 15th Int'l. Symposium on Computer Architecture* (Junho), 280-289.

Agarwal, A., J. Kubiatowicz, D. Kranz, B. -H. Lim, D. Yeung, G. D'Souza, e M. Parkin [1993]. "Sparcle: An evolutionary processor design for large-scale multiprocessors", *IEEE Micro* 13 (Junho), 48-61.

Agerwala, T., e J. Cocke [1987]. *High Performance Reduced Instruction Set Proces-sors*, IBM Tech. Rep. RC12434, IBM, Armonk, N.Y.

Akeley, K., e T. Jermoluk [1988]. "High-Performance Polygon Rendering", *Proc. 15th Annual Conf. on Computer Graphics and Interactive Techniques (SIGGRAPH 1988)*,1-5 de Agosto de 1988, Atlanta, Ga, 239-246.

Alexander, W. G., e D. B. Wortman [1975]. "Static and dynamic characteristics of XPL programs", *IEEE Computer* 8:11 (Novembro), 41-46.

Alles, A. [1995]. "ATM Internetworking", White Paper (Maio), Cisco Systems, Inc., San Jose, Calif.(*www.cisco.com/warp/public/614/12.html*)

Alliant. [1987]. *Alliant FX/Series: Product Summary*, Alliant Computer Systems Corp., Acton, Mass.

Almasi, G. S., e A. Gottlieb [1989]. *Highly Parallel Computing*, Benjamin/Cummings, Redwood City, Calif.

Alverson, G., R. Alverson, D. Callahan, B. Koblenz, A. Porterfield, e B. Smith [1992]. "Exploiting heterogeneous parallelism on a multithreaded multiprocessor", *Proc. ACM/IEEE Conf. on Supercomputing*, 16-20 de Novembro de 1992, Minneapolis, Minn., 188-197.

Amdahl, G. M. [1967]. "Validity of the single processor approach to achieving large scale computing capabilities", *Proc. AFIPS Spring Joint Computer Conf.*, 18-20 de Abril de 1967, Atlantic City, N.J., 483-485.

Amdahl, G. M., G. A. Blaauw, e F. P. Brooks, Jr. [1964]. "Architecture of the IBM System 360", *IBM J. Research and Development* 8:2 (Abril), 87-101.

Amza, C., A. L. Cox, S. Dwarkadas, P. Keleher, H. Lu, R. Rajamony, W. Yu, e W. Zwaenepoel [1996]. "Treadmarks: Shared memory computing on networks of workstations", *IEEE Computer* 29:2 (Fevereiro), 18-28.

Anderson, D. [2003]. "You don't know jack about disks", *Queue* 1:4 (Junho), 20-30.

Anderson, D., J. Dykes, e E. Riedel [2003]. "SCSI vs. ATA—More than an interface", *Proc. 2nd USENIX Conf. on File and Storage Technology (FAST '03)*31 de Março - 2 *de* Abril de 2003, San Francisco.

Anderson, D. W., F. J. Sparacio, e R. M. Tomasulo [1967]. "The IBM 360 Model 91: Processor philosophy and instruction handling", *IBM J. Research and Development* 11:1 (Janeiro), 8-24.

Anderson, M. H. [1990]. "Strength (and safety) in numbers (RAID, disk storage technol-ogy)", *Byte* 15:13 (Dezembro), 337-339.

Anderson, T. E., D. E. Culler, e D. Patterson [1995]. "A case for NOW (networks of workstations)", *IEEE Micro* 15:1 (Fevereiro), 54-64.

Ang, B., D. Chiou, D. Rosenband, M. Ehrlich, L. Rudolph, e Arvind [1998]. "StarT-Voyager: A flexible platform for exploring scalable SMP issues", *Proc. ACM/IEEE Conf. on Supercomputing*, 7-13 de Novembro de 1998, Orlando, FL.

Anjan, K. V., e T. M. Pinkston [1995]. "An efficient, fully-adaptive deadlock recovery scheme: Disha", *Proc. 22nd Annual Int'l. Symposium on Computer Architecture (ISCA)*22-24 de Junho de 1995, Santa Margherita, Itália, .

Anon. et al. [1985]. *A Measure of Transaction Processing Power*, Tandem Tech. Rep. TR85.2. Também aparece em *Datamation* 31:7 (Abril), 112-118, 1985.

Apache Hadoop. [2011]. *http://hadoop.apache.org*.

Archibald, J., e J. -L. Baer [1986]. "Cache coherence protocols: Evaluation using a multiprocessor simulation model", *ACM Trans. on Computer Systems* 4:4 (Novembro), 273-298.

Armbrust, M., A. Fox, R. Griffith, A. D. Joseph, R. Katz, A. Konwinski, G. Lee, D. Patter-son, A. Rabkin, I. Stoica, e M. Zaharia [2009]. *Above the Clouds: A Berkeley View of Cloud Computing*, Tech. Rep. UCB/EECS-2009-28, University of California, Berkeley (*http://www.eecs.berkeley.edu/Pubs/TechRpts/2009/EECS-2009-28.html*).

Arpaci, R. H., D. E. Culler, A. Krishnamurthy, S. G. Steinberg, e K. Yelick [1995]. "Empirical evaluation of the CRAY-T3D: A compiler perspective", *22nd Annual Int'l. Symposium on Computer Architecture (ISCA)*, 22-24 de Junho de 1995, Santa Margherita, Itália.

Asanovic, K. [1998]. "Vector Microprocessors", Ph.D. thesis, Computer Science Division, University of California, Berkeley.

Associated Press. [2005]. "Gap Inc. shuts down two Internet stores for major overhaul", *USATODAY.com*, Agosto 8, 2005.

Atanasoff, J. V. [1940]. *Computing Machine for the Solution of Large Systems of Linear Equations*, Internal Report, Iowa State University, Ames.

Atkins, M. [1991]. Performance and the i860 Microprocessor, *IEEE Micro* 11:5 (Setembro), 24-2772-78.

Austin, T. M., e G. Sohi [1992]. "Dynamic dependency analysis of ordinary programs", *Proc. 19th Annual Int'l. Symposium on Computer Architecture (ISCA)*19-21 de Maio de 1992, Gold Coast, Australia, 342-351.

Babbay, F., e A. Mendelson [1998]. Using value prediction to increase the power of speculative execution hardware", *ACM Trans. on Computer Systems* 16:3 (Agosto), 234-270.

Baer, J. -L., e W. -H. Wang [1988]. "On the inclusion property for multi-level cache hierarchies", *Proc. 15th Annual Int'l. Symposium on Computer Architecture*, 30 de Maio - 2 de Junho de 1988, Honolulu, Hawaii, 73-80.

Bailey, D. H., E. Barszcz, J. T. Barton, D. S. Browning, R. L. Carter, L. Dagum, R. A. Fatoohi, P. O. Frederickson, T. A. Lasinski, R. S. Schreiber, H. D. Simon, V. Venkatakrishnan, e S. K. Weeratunga [1991]. "The NAS parallel benchmarks", *Int'l. J. Supercomputing Applications* 5, 63-73.

Bakoglu, H. B., G. F. Grohoski, L. E. Thatcher, J. A. Kaeli, C. R. Moore, D. P. Tattle, W. E. Male, W. R. Hardell, D. A. Hicks, M. Nguyen Phu, R. K. Montoye, W. T. Glover, e S. Dhawan [1989]. "IBM second-generation RISC processor organization", *Proc. IEEE Int'l. Conf. on Computer Design*, 30 de Setembro - 4 de Outubro de 1989, Rye, N.Y., 138-142.

Balakrishnan, H., V. N. Padmanabhan, S. Seshan, e R. H. Katz [1997]. "A comparison of mechanisms for improving TCP performance over wireless links", *IEEE/ACM Trans. on Networking* 5:6 (Dezembro), 756-769.

Ball, T., e J. Larus [1993]. *"Branch prediction for free", Proc. ACM SIGPLAN'93 Con-ference on Programming Language Design and Implementation (PLDI)*, 23-25 de Junho de 1993, Albuquerque, N.M.,300-313.

Banerjee, U. [1979]. "Speedup of Ordinary Programs", Ph.D. thesis, Dept. of Computer Science, University of Illinois at Urbana-Champaign.

Barham, P., B. Dragovic, K. Fraser, S. Hand, T. Harris, A. Ho, e R. Neugebauer [2003]. "Xen and the art of virtualization", *Proc. of the 19th ACM Symposium on Operating Systems Principles*, 19-22 de Outubro de 2003, Bolton Landing, N.Y.

Barroso, L. A. [2010]. "Warehouse Scale Computing [keynote address]", *Proc. ACM SIGMOD*, 8-10 de Junho de 2010, Indianapolis, Ind.

Barroso, L. A., e U. Hölzle [2007]. The case for energy-proportional computing", *IEEE Computer* 40:12 (Dezembro), 33-37.

Barroso, L. A., e U. Hölzle [2009]. *The Datacenter as a Computer: An Introduction to the Design of Warehouse-Scale Machines*, Morgan & Claypool, San Rafael, Calif.

Barroso, L. A., K. Gharachorloo, e E. Bugnion [1998]. "Memory system characteriza-tion of commercial workloads", *Proc. 25th Annual Int'l. Symposium on Computer Architecture (ISCA)*, 3-14 de Julho de 1998, Barcelona, Espanha3-14.

Barton, R. S. [1961]. "A new approach to the functional design of a computer", *Proc. Western Joint Computer Conf.*, 9-11 de Maio de 1961, Los Angeles, Calif., 393-396.

Bashe, C. J., W. Buchholz, G. V. Hawkins, J. L. Ingram, e N. Rochester [1981]. "The architecture of IBM's early computers", *IBM J. Research and Development* 25:5 (Setembro), 363-375.

Bashe, C. J., L. R. Johnson, J. H. Palmer, e E. W. Pugh [1986]. *IBM's Early Computers*, MIT Press, Cambridge, Mass.

Baskett, F., e T. W. Keller [1977]. "An evaluation of the Cray-1 processor", *High Speed Computer and Algorithm Organization*, D. J. Kuck, D. H. Lawrie, e A. H. Sameh, eds., Academic Press,San Diego, 71-84.

Baskett, F., T. Jermoluk, e D. Solomon [1988]. "The 4D-MP graphics superworkstation: Computing+graphics=40 MIPS+40 MFLOPS and 10,000 lighted polygons per sec-ond", Proc. IEEE COMPCON, 29 de Fevereiro - 4 de Março de 1988, *San Francisco*, 468-471.

BBN Laboratories. [1986]. *Butterfly Parallel Processor Overview*, Tech. Rep. 6148, BBN Laboratories, Cambridge, Mass.

Bell, C. G. [1984]. "The mini and micro industries", *IEEE Computer* 17:10 (Outubro), 14-30.
Bell, C. G. [1985]. "Multis: A new class of multiprocessor computers", *Science* 228:(Abril 26), 462-467.
Bell, C. G. [1989]. "The future of high performance computers in science and engineer-ing", *Communications of the ACM* 32:9 (Setembro), 1091-1101.
Bell, G., e J. Gray [2001]. *Crays, Clusters and Centers*, Tech. Rep. MSR-TR-2001-76, Microsoft Research, Redmond, Wash.
Bell, C. G., e J. Gray [2002]. "What's next in high performance computing?", *CACM* 45:2 (Fevereiro), 91-95.
Bell, C. G., e A. Newell [1971]. *Computer Structures: Readings and Examples*, McGraw-Hill, New York.
Bell, C. G., e W. D. Strecker [1976]. "Computer structures: What have we learned from the PDP-11?", *Third Annual Int'l. Symposium on Computer Architecture (ISCA)*19-21 de Janeiro de 1976, Tampa, Fla., 1-14.
Bell, C. G., e W. D. Strecker [1998]. "Computer structures: What have we learned from the PDP-11?", *25 Years of the International Symposia on Computer Architecture (Selected Papers)*. ACM, New York, 138-151.
Bell, C. G., J. C. Mudge, e J. E. McNamara [1978]. *A DEC View of Computer Engineer-ing*, Digital Press, Bedford, Mass .
Bell, C. G., R. Cady, H. McFarland, B. DeLagi, J. O'Laughlin, R. Noonan, e W. Wulf [1970]. "A new architecture for mini-computers: The DEC PDP-11", *Proc. AFIPS Spring Joint Computer Conf.*, 5-7 de Maio de 1970, Atlantic City, N.J., 657-675.
Benes, V. E. [1962]. "Rearrangeable three stage connecting networks", *Bell System Tech-nical Journal* 41, 1481-1492.
Bertozzi, D., A. Jalabert, S. Murali, R. Tamhankar, S. Stergiou, L. Benini, e G. De Micheli [2005]. "NoC synthesis flow for customized domain specific multiprocessor systems-on-chip", *IEEE Trans. on Parallel and Distributed Systems* 16:2 (Fevereiro), 113-130.
Bhandarkar, D. P. [1995]. *Alpha Architecture and Implementations*, Digital Press: Newton, Mass.
Bhandarkar, D. P., e D. W. Clark [1991]. "Performance from architecture: Comparing a RISC and a CISC with similar hardware organizations", *Proc. Fourth Int'l. Conf. on Architectural Support for Programming Languages and Operating Systems (ASPLOS)*, 8-11 de Abril de 1991, Palo Alto, Calif., 310-319.
Bhandarkar, D. P., e J. Ding [1997]. "Performance characterization of the Pentium Pro processor", *Proc. Third Int'l. Symposium on High-Performance Computer Architec-ture*, 1-5 de Fevereiro de 1997, San Antonio, Tex., 288-297.
Bhuyan, L. N., e D. P. Agrawal [1984]. "Generalized hypercube and hyperbus struc-tures for a computer network", *IEEE Trans. on Computers* 32:4 (Abril), 322-333.
Bienia, C., S. Kumar, P. S. Jaswinder, e K. Li [2008]. *The Parsec Benchmark Suite: Characterization and Architectural Implications*, Tech. Rep. TR-811-08, Princeton University, Princeton, N.J.
Bier, J. [1997]. "The Evolution of DSP Processors", presentation at Univesity of California, Berkeley, 14 de Novembro.
Bird, S., A. Phansalkar, L. K. John, A. Mericas, e R. Indukuru [2007]. "Characterization of performance of SPEC CPU benchmarks on Intel's Core Microarchitecture based processor", *Proc. 2007 SPEC Benchmark Workshop*, Janeiro 21, 2007, Austin, Tex.
Birman, M., A. Samuels, G. Chu, T. Chuk, L. Hu, J. McLeod, e J. Barnes [1990]. "Developing the WRL3170/3171 SPARC floating-point coprocessors", *IEEE Micro* 10:1, 55-64.
Blackburn, M., R. Garner, C. Hoffman, A. M. Khan, K. S. McKinley, R. Bentzur, A. Diwan, D. Feinberg, D. Frampton, S. Z. Guyer, M. Hirzel, A. Hosking, M. Jump, H. Lee, J. E. B. Moss, A. Phansalkar, D. Stefanovic, T. VanDrunen, D. von Dincklage, e B. Wiedermann [2006]. "The DaCapo benchmarks: Java benchmarking develop-ment and analysis", *ACM SIGPLAN Conference on Object-Oriented Programming, Systems, Languages, and Applications (OOPSLA)*, 22-26 de Outubro de 2006169-190.
Blaum, M., J. Bruck, e A. Vardy [1996]. "MDS array codes with independent parity symbols", *IEEE Trans. on Information Theory* IT-42:Março, 529-542.
Blaum, M., J. Brady, J. Bruck, e J. Menon [1994]. "EVENODD: An optimal scheme for tolerating double disk failures in RAID architectures", *Proc. 21st Annual Int'l. Sym-posium on Computer Architecture (ISCA)*18-21 de Abril de 1994, Chicago, 245-254.
Blaum, M., J. Brady, J. Bruck, e J. Menon [1995]. "EVENODD: An optimal scheme for tolerating double disk failures in RAID architectures", *IEEE Trans. on Computers* 44:2 (Fevereiro), 192-202.
Blaum, M., J. Brady, J. Bruck, J. Menon, e A. Vardy [2001]. "The EVENODD code and its generalization", in H. Jin, T. Cortes, e R. Buyya, eds., *High Performance Mass Storage and Parallel I/O: Technologies and Applications*, Wiley-IEEE, New York, 187-208.
Bloch, E. [1959]. "The engineering design of the Stretch computer", *1959 Proceedings of the Eastern Joint Computer Conf.*, Dezembro 1-3, 1959, Boston, Mass., 48-59.
Boddie, J. R. [2000]. "History of DSPs", *www.lucent.com/micro/dsp/dsphist.html*.
Bolt, K. M. [2005]. "Amazon sees sales rise, profit fall", *Seattle Post-Intelligencer*, 25 de Outubro (http://seattlepi.nwsource.com/business/245943_techearns26.html).
Bordawekar, R., U. Bondhugula, e R. Rao [2010]. "Believe It or Not!: Multi-core CPUs can Match GPU Performance for a FLOP-Intensive Application!", *19th International Con-ference on Parallel Architecture and Compilation Techniques (PACT 2010)*, Vienna, Austria, 11-15 de Setembro de 2010, 537-538.
Borg, A., R. E. Kessler, e D. W. Wall [1990]. "Generation and analysis of very long address traces", *19th Annual Int'l. Symposium on Computer Architecture (ISCA)*19-21 de Maio de 1992, Gold Coast, Australia, 270-279.

Bouknight, W. J., S. A. Deneberg, D. E. McIntyre, J. M. Randall, A. H. Sameh, e D. L. Slotnick [1972]. "The Illiac IV system", *Proc. IEEE* 60:4, 369-379. Também aparece em D. P. Siewiorek, C. G. Bell, e A. Newell, *Computer Structures: Principles and Examples*, McGraw-Hill, New York, 1982, 306-316.

Brady, J. T. [1986]. "A theory of productivity in the creative process", *IEEE CG&A*, Maio, 25-34.

Brain, M. [2000]. "Inside a Digital Cell Phone", *www.howstuffworks.com/inside-cellphone.htm*.

Brandt, M., J. Brooks, M. Cahir, T. Hewitt, E. Lopez-Pineda, e D. Sandness [2000]. *The Benchmarker's Guide for Cray SV1 Systems*, Cray Inc., Seattle, Wash.

Brent, R. P., e H. T. Kung [1982]. "A regular layout for parallel adders", *IEEE Trans. on Computers* C-31, 260-264.

Brewer, E. A., e B. C. Kuszmaul [1994]. "How to get good performance from the CM-5 data network", *Proc. Eighth Int'l. Parallel Processing Symposium*, 26-27 de Abril de 1994, Cancun, Mexico.

Brin, S., e L. Page [1998]. "The anatomy of a large-scale hypertextual Web search engine", *Proc. 7th Int'l. World Wide Web Conf.*, 14-18 de Abril de 1998, Brisbane, Queensland, Australia, 107-117.

Brown, A., e D. A. Patterson [2000]. "Towards maintainability, availability, and growth benchmarks: A case study of software RAID systems", *Proc. 2000 USENIX Annual Technical Conf* 18-23 de Junho de 2000, San Diego, Calif, .

Bucher, I. V., e A. H. Hayes [1980]. "I/O performance measurement on Cray-1 and CDC 7000 computers", *Proc. Computer Performance Evaluation Users Group, 16th Meeting*, NBS 500-65, 245-254.

Bucher, I. Y. [1983]. "The computational speed of supercomputers", *Proc. Int'l. Conf. on Measuring and Modeling of Computer Systems (SIGMETRICS 1983)*, 29-31 de Agosto de 1983, Minneapolis, Minn., 151-165.

Bucholtz, W. [1962]. *Planning a Computer System: Project Stretch*, McGraw-Hill, New York .

Burgess, N., e T. Williams [1995]. Choices of operand truncation in the SRT division algorithm", *IEEE Trans. on Computers* 44:7, 933-938.

Burkhardt III, H., S. Frank, B. Knobe, e J. Rothnie [1992]. *Overview of the KSR1 Com-puter System*, Tech. Rep. KSR-TR-9202001, Kendall Square Research, Boston, Mass.

Burks, A. W., H. H. Goldstine, e J. von Neumann [1946]. "Preliminary discussion of the logical design of an electronic computing instrument", Report to the U.S. Army Ordnance Department, p. 1; also appears in *Papers of John von Neumann*, W. Aspray and A. Burks, eds., MIT Press, Cambridge, Mass., e Tomash Publishers, Los Angeles, Calif., 1987, 97-146.

Calder, B., G. Reinman, e D. M. Tullsen [1999]. "Selective value prediction", *Proc. 26th Annual Int'l. Symposium on Computer Architecture (ISCA)*, 2-4 de Maio de 1999, Atlanta, Ga.

Calder, B., D. Grunwald, M. Jones, D. Lindsay, J. Martin, M. Mozer, e B. Zorn [1997]. "Evidence-based static branch prediction using machine learning", *ACM Trans. Pro-gram. Lang. Syst.* 19:1, 188-222.

Callahan, D., J. Dongarra, e D. Levine [1988]. "Vectorizing compilers: A test suite and results", *Proc. ACM/IEEE Conf. on Supercomputing*, 12-17 de Novembro de 1988, Orland, Fla., 98-105.

Cantin, J. F., e M. D. Hill [2001]. "Cache Performance for Selected SPEC CPU2000 Benchmarks", *www.jfred.org/cache-data.html* (Junho).

Cantin, J. F., e M. D. Hill [2003]. "Cache Performance for SPEC CPU2000 Benchmarks, Version 3.0", *www.cs.wisc.edu/multifacet/misc/spec2000cache-data/index.html*.

Carles, S. [2005]. "Amazon reports record Xmas season, top game picks", *Gamasutra*, Dezembro 27 (*http://www.gamasutra.com/php-bin/news_index.php?story=7630*.)

Carter, J., e K. Rajamani [2010]. "Designing energy-efficient servers and data centers", *IEEE Computer* 43:7 (Julho), 76-78.

Case, R. P., e A. Padegs [1978]. "The architecture of the IBM System/370", *Communications of the ACM* 21:1, 73-96. Também aparece em D. P. Siewiorek, C. G. Bell, and A. Newell, *Computer Structures: Principles and Examples*, McGraw-Hill, New York, 1982, 830-855.

Censier, L., e P. Feautrier [1978]. "A new solution to coherence problems in multicache systems", *IEEE Trans. on Computers* C-27:12 (Dezembro), 1112-1118.

Chandra, R., S. Devine, B. Verghese, A. Gupta, e M. Rosenblum [1994]. "Scheduling and page migration for multiprocessor compute servers", *Sixth Int'l. Conf. on Archi-tectural Support for Programming Languages and Operating Systems (ASPLOS)*, 4-7 de Outubro de 1994, San Jose, Calif., 12-24.

Chang, F., J. Dean, S. Ghemawat, W. C. Hsieh, D. A. Wallach, M. Burrows, T. Chandra, A. Fikes, e R. E. Gruber [2006]. "Bigtable: A distributed storage system for struc-tured data", *Proc. 7th USENIX Symposium on Operating Systems Design and Imple-mentation (OSDI '06)*, 6-8 de Novembro de 2006, Seattle, Wash.

Chang, J., J. Meza, P. Ranganathan, C. Bash, e A. Shah [2010]. "Green server design: Beyond operational energy to sustainability", *Proc. Workshop on Power Aware Com-puting and Systems (HotPower '10)*, 3 de Outubro de 2010, Vancouver, British Columbia.

Chang, P. P., S. A. Mahlke, W. Y. Chen, N. J. Warter, e W. W. Hwu [1991]. "IMPACT: An architectural framework for multiple-instruction-issue processors", *18th Annual Int'l. Symposium on Computer Architecture (ISCA)*, 27-30 de Maio de 1991, Toronto, Canadá, 266-275.

Charlesworth, A. E. [1981]. "An approach to scientific array processing: The architecture design of the AP-120B/FPS-164 family", *Computer* 14:9 (Setembro), 18-27.

Charlesworth, A. [1998]. "Starfire: Extending the SMP envelope", *IEEE Micro* 18:1 (Janeiro/Fevereiro), 39-49.

Chen, P. M., e E. K. Lee [1995]. "Striping in a RAID level 5 disk array", *Proc. ACM SIGMETRICS Conf. on Measurement and Modeling of Computer Systems*, 15-19 de Maio de 1995, Ottawa, Canadá, 136-145.

Chen, P. M., G. A. Gibson, R. H. Katz, e D. A. Patterson [1990]. "An evaluation of redundant arrays of inexpensive disks using an Amdahl 5890", *Proc. ACM SIGMET-RICS Conf. on Measurement and Modeling of Computer Systems*, 22-25 de Maio de 1990, Boulder, Colo.

Chen, P. M., E. K. Lee, G. A. Gibson, R. H. Katz, e D. A. Patterson [1994]. "RAID: High-performance, reliable secondary storage", *ACM Computing Surveys* 26:2 (Junho), 145-188.

Chen, S. [1983]. "Large-scale and high-speed multiprocessor system for scientific applications", *Proc. NATO Advanced Research Workshop on High-Speed Computing*, 20-22 de Junho de 1983, Jülich, Alemanha Ocidental. Também aparece em K. Hwang, ed., "Superprocessors: Design and applications", *IEEE* (Agosto), 602-609, 1984.

Chen, T. C. [1980]. "Overlap and parallel processing", in H. Stone, ed., *Introduction to Computer Architecture*, Science Research Associates, Chicago, 427-486.

Chow, F. C. [1983]. "A Portable Machine-Independent Global Optimizer—Design and Measurements", Tese de doutorado, Stanford University, Palo Alto, Calif.

Chrysos, G. Z., e J. S. Emer [1998]. "Memory dependence prediction using store sets", *Proc. 25th Annual Int'l. Symposium on Computer Architecture (ISCA)*, 3-14 de Julho de 1998, Barcelona, Espanha, 142-153.

Clark, B., T. Deshane, E. Dow, S. Evanchik, M. Finlayson, J. Herne, e J. Neefe Matthews [2004]. "Xen and the art of repeated research", *Proc. USENIX Annual Technical Conf.*, 27 de Junho - 2 de Julho de 2004, 135-144.

Clark, D. W. [1983]. "Cache performance of the VAX-11/780", *ACM Trans. on Computer Systems* 1:1, 24-37.

Clark, D. W. [1987]. "Pipelining and performance in the VAX 8800 processor", *Proc. Second Int'l. Conf. on Architectural Support for Programming Languages and Operating Systems (ASPLOS)*, 5-8 de Outubro de 1987, Palo Alto, Calif., 173-177.

Clark, D. W., e J. S. Emer [1985]. "Performance of the VAX-11/780 translation buffer: Simulation and measurement", *ACM Trans. on Computer Systems*, 3:1 (Fevereiro), 31-62.

Clark, D., e H. Levy [1982]. "Measurement and analysis of instruction set use in the VAX-11/780", *Proc. Ninth Annual Int'l. Symposium on Computer Architecture (ISCA)*, 26-29 de Abril de 1982, Austin, Tex., 9-17.

Clark, D., e W. D. Strecker [1980]. "Comments on 'the case for the reduced instruction set computer', *Computer Architecture News*, 8:6 (Outubro), 34-38.

Clark, W. A. [1957]. "The Lincoln TX-2 computer development", *Proc. Western Joint Computer Conference,* 26-28 de Fevereiro de 1957, Los Angeles, 143-145.

Clidaras, J., C. Johnson, e B. Felderman [2010]. *Private communication*.

Climate Savers Computing Initiative. [2007]. "Efficiency Specs", *http://www.climatesaverscomputing.org/*.

Clos, C. [1953]. "A study of non-blocking switching networks", *Bell Systems Technical Journal*, 32:Março, 406-424.

Cody, W. J., J. T. Coonen, D. M. Gay, K. Hanson, D. Hough, W. Kahan, R. Karpinski, J. Palmer, F. N. Ris, e D. Stevenson [1984]. "A proposed radix- and word-lengthindependent standard for floating-point arithmetic", *IEEE Micro* 4:4, 86-100.

Colwell, R. P., e R. Steck [1995]. "A 0.6 μm BiCMOS processor with dynamic exe-cution", *Proc. of IEEE Int'l. Symposium on Solid State Circuits (ISSCC)*, 15-17 de Fevereiro de 1995, San Francisco, 176-177.

Colwell, R. P., R. P. Nix, J. J. O'Donnell, D. B. Papworth, e P. K. Rodman [1987]. "A VLIW architecture for a trace scheduling compiler", *Proc. Second Int'l. Conf. on Architectural Support for Programming Languages and Operating Systems (ASPLOS)*, 5-8 de Outubro de 1987, Palo Alto, Calif.180-192.

Comer, D. [1993]. *Internetworking with TCP/IP*, 2nd ed., Prentice Hall, Englewood Cliffs, N.J.

Compaq Computer Corporation. [1999]. *Compiler Writer's Guide for the Alpha 21264*, Order Number EC-RJ66A-TE, Junho, *www1.support.compaq.com/alpha-tools/documentation/current/21264_EV67/ec-rj66a-te_comp_writ_gde_for_alpha21264.pdf*.

Conti, C., D. H. Gibson, e S. H. Pitkowsky [1968]. "Structural aspects of the System/360 Model 85. Part I. General organization", *IBM Systems J* 7:1, 2-14.

Coonen, J. [1984]. "Contributions to a Proposed Standard for Binary Floating-Point Arithmetic", Ph.D. thesis, University of California, Berkeley.

Corbett, P., B. English, A. Goel, T. Grcanac, S. Kleiman, J. Leong, e S. Sankar [2004]. "Row-diagonal parity for double disk failure correction", *Proc. 3rd USENIX Conf. on File and Storage Technology (FAST '04)*, Março 31-Abril 2, 2004, San Francisco.

Crawford, J., e P. Gelsinger [1988]. *Programming the 80386*, Sybex Books, Alameda, Calif.

Culler, D. E., J. P. Singh, e A. Gupta [1999]. *Parallel Computer Architecture: A Hardware/Software Approach*, Morgan Kaufmann, San Francisco.

Curnow, H. J., e B. A. Wichmann [1976]. "A synthetic benchmark", *The Computer J* 19:1, 43-49.

Cvetanovic, Z., e R. E. Kessler [2000]. "Performance analysis of the Alpha 21264-based Compaq ES40 system", *Proc. 27th Annual Int'l. Symposium on Computer Architecture (ISCA)*, 10-14 de Junho de 2000, Vancouver, Canadá, 192-202.

Dally, W. J. [1990]. "Performance analysis of *k*-ary *n*-cube interconnection networks", *IEEE Trans. on Computers,* 39:6 (Junho), 775-785.

Dally, W. J. [1992]. "Virtual channel flow control", *IEEE Trans. on Parallel and Distrib-uted Systems*, 3:2 (Março), 194-205.

Dally, W. J. [1999]. "Interconnect limited VLSI architecture", *Proc. of the International Interconnect Technology Conference*, 24-26 de Maio de 1999, San Francisco.

Dally, W. J., e C. I. Seitz [1986]. "The torus routing chip", *Distributed Computing* 1:4, 187-196.
Dally, W. J., e B. Towles [2001]. "Route packets, not wires: On-chip interconnection networks", *Proc. 38th Design Automation Conference*, 18-22 de Junho de 2001, Las Vegas.
Dally, W. J., e B. Towles [2003]. *Principles and Practices of Interconnection Networks*, Morgan Kaufmann, San Francisco.
Darcy, J. D., e D. Gay [1996]. "FLECKmarks: Measuring floating point performance using a full IEEE compliant arithmetic benchmark", CS 252 class project, University of California, Berkeley (veja *HTTP.CS.Berkeley.EDU/~darcy/Projects/cs252/*).
Darley, H. M. et al. [1989]. "Floating Point/Integer Processor with Divide and Square Root Functions", U.S. Patent 4,878,190, 31 de Outubro.
Davidson, E. S. [1971]. "The design and control of pipelined function generators", *Proc. IEEE Conf. on Systems, Networks, and Computers*, 19—21 de Janeiro de 1971, Oaxtepec, Mexico, 19-21.
Davidson, E. S., A. T. Thomas, L. E. Shar, e J. H. Patel [1975]. "Effective control for pipelined processors", *Proc. IEEE COMPCON*, 25-27 de Fevereiro de 1975, San Francisco, 181-184.
Davie, B. S., L. L. Peterson, e D. Clark [1999]. *Computer Networks: A Systems Approach*, 2nd ed., Morgan Kaufmann, San Francisco.
Dean, J. [2009]. Designs lessons and advice from building large distributed systems [keynote address]", *Proc. 3rd ACM SIGOPS Int'l. Workshop on Large-Scale Distributed Systems and Middleware, Co-located with the 22nd ACM Symposium on Operating Systems Principles*, 11-14 Outubro de 2009, Big Sky, Mont.
Dean, J., e S. Ghemawat [2004]. "MapReduce: Simplified data processing on large clusters", *Proc. Operating Systems Design and Implementation (OSDI)*, 6-8 de Dezembro de 2004, San Francisco, Calif, 137-150.
Dean, J., e S. Ghemawat [2008]. "MapReduce: Simplified data processing on large clusters", *Communications of the ACM* 51:1, 107-113.
DeCandia, G., D. Hastorun, M. Jampani, G. Kakulapati, A. Lakshman, A. Pilchin, S. Sivasubramanian, P. Vosshall, e W. Vogels [2007]. "Dynamo: Amazon's highly available key-value store", *Proc. 21st ACM Symposium on Operating Systems Principles*, 14-17 de Outubro de 2007, Stevenson, Wash.
Dehnert, J. C., P. Y. -T. Hsu, e J. P. Bratt [1989]. "Overlapped loop support on the Cydra 5", *Proc. Third Int'l. Conf. on Architectural Support for Programming Lan-guages and Operating Systems (ASPLOS)*, 3-6 de Abril de 1989, Boston, Mass., 26-39.
Demmel, J. W., e X. Li [1994]. "Faster numerical algorithms via exception handling", *IEEE Trans. on Computers* 43:8, 983-992.
Denehy, T. E., J. Bent, F. I. Popovici, A. C. Arpaci-Dusseau, e R. H. Arpaci-Dusseau [2004]. "Deconstructing storage arrays", *Proc. 11th Int'l. Conf. on Architectural Sup-port for Programming Languages and Operating Systems (ASPLOS)*, 7-13 de Outubro de 2004, Boston, Mass, 59-71.
Desurvire, E. [1992]. "Lightwave communications: The fifth generation", *Scientific American* (International Edition), 266:1 (Janeiro), 96-103.
Diep, T. A., C. Nelson, e J. P. Shen [1995]. "Performance evaluation of the PowerPC 620 microarchitecture", *Proc. 22nd Annual Int'l. Symposium on Computer Architec-ture (ISCA)*, 22-24 de Junho 22-24 de Santa Margherita, Itália.
Digital Semiconductor [1996]. *Alpha Architecture Handbook, Version 3*, Digital Press: Maynard, Mass.
Ditzel, D. R., e H. R. McLellan [1987]. "Branch folding in the CRISP microprocessor: Reducing the branch delay to zero", *Proc. 14th Annual Int'l. Symposium on Computer Architecture (ISCA)*, 2-5 de Junho de 1987, Pittsburgh, Penn., 2-7.
Ditzel, D. R., e D. A. Patterson [1980]. "Retrospective on high-level language computer architecture", *Proc. Seventh Annual Int'l. Symposium on Computer Architecture (ISCA)*, 6-8 de Maio de 1980, La Baule, França, 97-104.
Doherty, W. J., e R. P. Kelisky [1979]. "Managing VM/CMS systems for user effec-tiveness", *IBM Systems J* 18:1, 143-166.
Dongarra, J. J. [1986]. "A survey of high performance processors", *Proc. IEEE COMPCON*, 3-6 de Março de 1986, San Francisco, 8-11.
Dongarra, J., T. Sterling, H. Simon, e E. Strohmaier [2005]. "High-performance computing: Clusters, constellations, MPPs, and future directions", *Computing in Science & Engineering*, 7:2 (Março/Abril), 51-59.
Douceur, J. R., e W. J. Bolosky [1999]. "A large scale study of file-system contents", *Proc. ACM SIGMETRICS Conf. on Measurement and Modeling of Computer Systems*, 1-9 de Maio de 1999, Atlanta, Ga., 59-69.
Douglas, J. [2005]. "Intel 8xx series and Paxville Xeon-MP microprocessors", paper pre-sented at Hot Chips 17, Agosto 14-16, 2005, Stanford University, Palo Alto, Calif.
Duato, J. [1993]. "A new theory of deadlock-free adaptive routing in wormhole networks", *IEEE Trans. on Parallel and Distributed Systems*, 4:12 (Dezembro), 1320-1331.
Duato, J., e T. M. Pinkston [2001]. "A general theory for deadlock-free adaptive routing using a mixed set of resources", *IEEE Trans. on Parallel and Distributed Systems*, 12:12 (Dezembro), 1219-1235.
Duato, J., S. Yalamanchili, e L. Ni [2003]. *Interconnection Networks: An Engineering Approach*, 2nd printing, Morgan Kaufmann, San Francisco.
Duato, J., I. Johnson, J. Flich, F. Naven, P. Garcia, e T. Nachiondo [2005a]. "A new scalable and cost-effective congestion management strategy for lossless multistage interconnection networks", *Proc. 11th Int'l. Symposium on High-Performance Computer Architecture*, 12-16 de Fevereiro de 2005, San Francisco.

Duato, J., O. Lysne, R. Pang, e T. M. Pinkston [2005b]. "Part I: A theory for deadlock-free dynamic reconfiguration of interconnection networks", *IEEE Trans. on Parallel and Distributed Systems*, 16:5 (Maio), 412-427.

Dubois, M., C. Scheurich, e F. Briggs [1988]. "Synchronization, coherence, and event ordering", *IEEE Computer*, 21:2 (Fevereiro), 9-21.

Dunigan, W., K. Vetter, K. White, e P. Worley [2005]. "Performance evaluation of the Cray X1 distributed shared memory architecture", *IEEE Micro*, Janeiro/Fevereiro, 30-40.

Eden, A., e T. Mudge [1998]. "The YAGS branch prediction scheme", *Proc. of the 31st Annual ACM/IEEE Int'l. Symposium on Microarchitecture*, 30 de Novembro - 2 de Dezembro de 1998, Dallas, Tex., 69-80.

Edmondson, J. H., P. I. Rubinfield, R. Preston, e V. Rajagopalan [1995]. "Superscalar instruction execution in the 21164 Alpha microprocessor", *IEEE Micro* 15:2, 33-43.

Eggers, S. [1989]. "Simulation Analysis of Data Sharing in Shared Memory Multiproces-sors", Ph.D. thesis, University of California, Berkeley.

Elder, J., A. Gottlieb, C. K. Kruskal, K. P. McAuliffe, L. Randolph, M. Snir, P. Teller, e J. Wilson [1985]. "Issues related to MIMD shared-memory computers: The NYU Ultracomputer approach", *Proc. 12th Annual Int'l. Symposium on Computer Architec-ture (ISCA)*, 17-19 de Junho de 1985, Boston, Mass, 126-135.

Ellis, J. R. [1986]. *Bulldog: A Compiler for VLIW Architectures*, MIT Press, Cambridge, Mass.

Emer, J. S., e D. W. Clark [1984]. "A characterization of processor performance in the VAX-11/780", *Proc. 11th Annual Int'l. Symposium on Computer Architecture (ISCA)*, 5-7 de Junho de 1984, Ann Arbor, Mich., 301-310.

Enriquez, P. [2001]. "What happened to my dial tone? A study of FCC service disruption reports", poster, *Richard Tapia Symposium on the Celebration of Diversity in Com-puting*, Outubro 18-20, Houston, Tex.

Erlichson, A., N. Nuckolls, G. Chesson, e J. L. Hennessy [1996]. "SoftFLASH: Analyz-ing the performance of clustered distributed virtual shared memory", *Proc. Seventh Int'l. Conf. on Architectural Support for Programming Languages and Operating Systems (ASPLOS)*, 1-5 de Outubro de 1996, Cambridge, Mass., 210-220.

Esmaeilzadeh, H., T. Cao, Y. Xi, S. M. Blackburn, e K. S. McKinley [2011]. "Looking Back on the Language and Hardware Revolution: Measured Power, Performance, and Scaling", *Proc. 16th Int'l. Conf. on Architectural Support for Programming Lan-guages and Operating Systems (ASPLOS)*, 5-11 de Março de 2011, Newport Beach, Calif.

Evers, M., S. J. Patel, R. S. Chappell, e Y. N. Patt [1998]. "An analysis of correlation and predictability: What makes two-level branch predictors work", *Proc. 25th Annual Int'l. Symposium on Computer Architecture (ISCA)*, 3-14 de Julho de Barcelona, , *Espanha*, 52-61.

Fabry, R. S. [1974]. "Capability based addressing", *Communications of the ACM*, 17:7 (Julho), 403-412.

Falsafi, B., e D. A. Wood [1997]. "Reactive NUMA: A design for unifying S-COMA and CC-NUMA", *Proc. 24th Annual Int'l. Symposium on Computer Architecture (ISCA)*, 2-4 de Junho de 1997, Denver, Colo, 229-240.

Fan, X., W. Weber, e L. A. Barroso [2007]. "Power provisioning for a warehouse-sized computer", *Proc. 34th Annual Int'l. Symposium on Computer Architecture (ISCA)*, 9-13 de Junho de 2007, San Diego, Calif.

Farkas, K. I., e N. P. Jouppi [1994]. "Complexity/performance trade-offs with non-blocking loads", *Proc. 21st Annual Int'l. Symposium on Computer Architecture (ISCA)*, Abril 18-21, 1994, Chicago.

Farkas, K. I., N. P. Jouppi, e P. Chow [1995]. "How useful are non-blocking loads, stream buffers and speculative execution in multiple issue processors?", *Proc. First IEEE Symposium on High-Performance Computer Architecture*, 22-25 de Janeiro de 1995, Raleigh, N.C., 78-89.

Farkas, K. I., P. Chow, N. P. Jouppi, e Z. Vranesic [1997]. "Memory-system design considerations for dynamically-scheduled processors", *Proc. 24th Annual Int'l. Sympo-sium on Computer Architecture (ISCA)*, 2-4 de Junho de 1997, Denver, Colo., 133-143.

Fazio, D. [1987]. "It's really much more fun building a supercomputer than it is simply inventing one", *Proc. IEEE COMPCON*, 23-27 de Fevereiro de 1987, San Francisco, 102-105.

Fisher, J. A. [1981]. "Trace scheduling: A technique for global microcode compaction", *IEEE Trans. on Computers*, 30:7 (Julho), 478-490.

Fisher, J. A. [1983]. "Very long instruction word architectures and ELI-512", *10th Annual Int'l. Symposium on Computer Architecture (ISCA)*, 5-7 de Junho de 1982, Estocolmo, Suécia, 140-150.

Fisher, J. A., e S. M. Freudenberger [1992]. "Predicting conditional branches from pre-vious runs of a program", *Proc. Fifth Int'l. Conf. on Architectural Support for Programming Languages and Operating Systems (ASPLOS)*, 12-15 de Outubro de 1992, Boston, Mass., 85-95.

Fisher, J. A., e B. R. Rau [1993]. *Journal of Supercomputing*, Janeiro (edição especial).

Fisher, J. A., J. R. Ellis, J. C. Ruttenberg, e A. Nicolau [1984]. "Parallel processing: A smart compiler e a dumb processor", *Proc. SIGPLAN Conf. on Compiler Construction*, 17-22 de Junho de 1984, Montreal, Canadá, 11-16.

Flemming, P. J., e J. J. Wallace [1986]. "How not to lie with statistics: The correct way to summarize benchmarks results", *Communications of the ACM*, 29:3 (Março), 218-221.

Flynn, M. J. [1966]. "Very high-speed computing systems", *Proc. IEEE*, 54:12 (Decem-ber), 1901-1909.

Forgie, J. W. [1957]. "The Lincoln TX-2 input-output system", *Proc. Western Joint Com-puter Conference* (Fevereiro), Institute of Radio Engineers, Los Angeles, 156-160.

Foster, C. C., e E. M. Riseman [1972]. "Percolation of code to enhance parallel dispatch-ing and execution", *IEEE Trans. on Computers*, C-21:12 (Dezembro), 1411-1415.

Frank, S. J. [1984]. "Tightly coupled multiprocessor systems speed memory access time", *Electronics*, 57:1 (Janeiro), 164-169.

Freiman, C. V. [1961]. Statistical analysis of certain binary division algorithms", *Proc. IRE* 49:1, 91-103.

Friesenborg, S. E., e R. J. Wicks [1985]. *DASD Expectations: The 3380, 3380-23, and MVS/XA*, Tech. Bulletin GG22-9363-02, IBM Washington Systems Center, Gaithers-burg, Md.

Fuller, S. H., e W. E. Burr [1977]. "Measurement and evaluation of alternative com-puter architectures", *Computer*, 10:10 (Outubro), 24-35.

Furber, S. B. [1996]. *ARM System Architecture*, Addison-Wesley, Harlow, England (veja *www.cs.man.ac.uk/amulet/publications/books/ARMsysArch*).

Gagliardi, U. O. [1973]. "Report of workshop 4—software-related advances in computer hardware", *Proc. Symposium on the High Cost of Software*, 17-19 de Setembro de 1973, Monterey, Calif., 99-120.

Gajski, D., D. Kuck, D. Lawrie, e A. Sameh [1983]. "CEDAR—a large scale multipro-cessor", *Proc. Int'l. Conf. on Parallel Processing (ICPP)*, Agosto, Columbus, Ohio, 524-529.

Gallagher, D. M., W. Y. Chen, S. A. Mahlke, J. C. Gyllenhaal, e W. W. Hwu [1994]. "Dynamic memory disambiguation using the memory conflict buffer", *Proc. Sixth Int'l. Conf. on Architectural Support for Programming Languages and Operating Systems (ASPLOS)*, 4-7 de Outubro, San Jose, Calif, 183-193.

Galles, M. [1996]. "Scalable pipelined interconnect for distributed endpoint routing: The SGI SPIDER chip", *Proc. IEEE HOT Interconnects, 96*15-17 de Agosto de 1996, Stanford University, Palo Alto, Calif, .

Game, M., e A. Booker [1999]. "CodePack code compression for PowerPC proces-sors", *Micro*News, 5:1, *www.chips.ibm.com/micronews/vol5_no1/codepack.html*.

Gao, Q. S. [1993]. "The Chinese remainder theorem and the prime memory system", *20th Annual Int'l. Symposium on Computer Architecture (ISCA)*, 16-19 de Maio de 1993, San Diego, Calif. (*Computer Architecture News* 21:2 (Maio), 337-340).

Gap. [2005]. "Gap Inc. Reports Third Quarter Earnings", *http://gapinc.com/public/documents/PR_Q405Earnings-Feb2306.pdf*.

Gap. [2006]. "Gap Inc. Reports Fourth Quarter and Full Year Earnings", *http://gapinc.com/public/documents/Q32005PressRelease_Final22.pdff*.

Garner, R., A. Agarwal, F. Briggs, E. Brown, D. Hough, B. Joy, S. Kleiman, S. Muchnick, M. Namjoo, D. Patterson, J. Pendleton, e R. Tuck [1988]. "Scalable processor architecture (SPARC)", *Proc. IEEE COMPCON*, 29 de Fevereiro - 4 Março de 1988, San Francisco, 278-283.

Gebis, J., e D. Patterson [2007]. "Embracing and extending 20th-century instruction set architectures", *IEEE Computer*, 40:4 (Abril), 68-75.

Gee, J. D., M. D. Hill, D. N. Pnevmatikatos, e A. J. Smith [1993]. "Cache performance of the SPEC92 benchmark suite", *IEEE Micro* 13:4 (Agosto), 17-27.

Gehringer, E. F., D. P. Siewiorek, e Z. Segall [1987]. *Parallel Processing: The Cm* Experience*, Digital Press, Bedford, Mass.

Gharachorloo, K., A. Gupta, e J. L. Hennessy [1992]. "Hiding memory latency using dynamic scheduling in shared-memory multiprocessors", *Proc. 19th Annual Int'l. Sym-posium on Computer Architecture (ISCA)*, 19-21 de Maio de 1992, Gold Coast, Australia.

Gharachorloo, K., D. Lenoski, J. Laudon, P. Gibbons, A. Gupta, e J. L. Hennessy [1990]. "Memory consistency and event ordering in scalable shared-memory multi-processors", *Proc. 17th Annual Int'l. Symposium on Computer Architecture (ISCA)*, 28-31 de Maio de 1990, Seattle, Wash., 15-26.

Ghemawat, S., H. Gobioff, e S. -T. Leung [2003]. "The Google file system", *Proc. 19th ACM Symposium on Operating Systems Principles*, 19-22 de Outubro de 2003, Bolton Landing, N.Y..

Gibson, D. H. [1967]. "Considerations in block-oriented systems design", *AFIPS Conf. Proc.* 30, 75-80.

Gibson, G. A. [1992]. *Redundant Disk Arrays: Reliable, Parallel Secondary Storage*, ACM Distinguished Dissertation Series, MIT Press, Cambridge, Mass.

Gibson, J. C. [1970]. "The Gibson mix", Rep. TR. 00.2043, IBM Systems Development Division, Poughkeepsie, N.Y. (pesquisa realziada em 1959).

Gibson, J., R. Kunz, D. Ofelt, M. Horowitz, J. Hennessy, e M. Heinrich [2000]. "FLASH vs. (simulated) FLASH: Closing the simulation loop", *Proc. Ninth Int'l. Conf. on Architectural Support for Programming Languages and Operating Systems (ASPLOS)*, 12-15 de Novembro, Cambridge, Mass., 49-58.

Glass, C. J., e L. M. Ni [1992]. "The Turn Model for adaptive routing", *19th Annual Int'l. Symposium on Computer Architecture (ISCA)*, 19-21 de Maio de 1992, Gold Coast, Australia.

Goldberg, D. [1991]. "What every computer scientist should know about floating-point arithmetic", *Computing Surveys* 23:1, 5-48.

Goldberg, I. B. [1967]. "27 bits are not enough for 8-digit accuracy", *Communications of the ACM* 10:2, 105-106.

Goldstein, S. [1987]. *Storage Performance—An Eight Year Outlook*, Tech. Rep. TR 03.308-1, Santa Teresa Laboratory, IBM Santa Teresa Laboratory, San Jose, Calif.

Goldstine, H. H. [1972]. *The Computer: From Pascal to von Neumann*, Princeton Univer-sity Press: Princeton, N.J.

González, J., e A. González [1998]. "Limits of instruction level parallelism with data speculation", *Proc. Vector and Parallel Processing (VECPAR) Conf*, 21-23 de Junho de 1998, Porto, Portugal, 585-598.

Goodman, J. R. [1983]. "Using cache memory to reduce processor memory traffic", *Proc. 10th Annual Int'l. Symposium on Computer Architecture (ISCA)*, 5-7 de Junho de 1982, Estocolmo, Suécia, 124-131.

Goralski, W. [1997]. *SONET: A Guide to Synchronous Optical Network*, McGraw-Hill, New York.

Gosling, J. B. [1980]. *Design of Arithmetic Units for Digital Computers*, Springer-Verlag, New York.
Gray, J. [1990]. "A census of Tandem system availability between 1985 and 1990", *IEEE Trans. on Reliability*, 39:4 (Outubro), 409-418.
Gray J. (ed.) [1993]. *The Benchmark Handbook for Database and Transaction Process-ing Systems*, 2nd ed., Morgan Kaufmann, San Francisco.
Gray, J. [2006]. Sort benchmark home page, *http://sortbenchmark.org/*.
Gray, J., e A. Reuter [1993]. *Transaction Processing: Concepts and Techniques*, Morgan Kaufmann, San Francisco.
Gray, J., e D. P. Siewiorek [1991]. "High-availability computer systems", *Computer* 24:9 (Setembro), 39-48.
Gray, J., e C. van Ingen [2005]. *Empirical Measurements of Disk Failure Rates and Error Rates*, MSR-TR-2005-166, Microsoft Research, Redmond, Wash.
Greenberg, A., N. Jain, S. Kandula, C. Kim, P. Lahiri, D. Maltz, P. Patel, e S. Sengupta [2009]. "VL2: A Scalable and Flexible Data Center Network", in *Proc. ACM SIGCOMM*, 17-21 de Agosto de 2009, Barcelona, Espanha.
Grice, C., e M. Kanellos [2000]. "Cell phone industry at crossroads: Go high or low?", *CNET News, 31* de Agosto, *technews.netscape.com/news/0-1004-201-2518386-0.html?tag=st.ne.1002.tgif.sf*.
Groe, J. B., e L. E. Larson [2000]. *CDMA Mobile Radio Design*, Artech House, Boston.
Gunther, K. D. [1981]. "Prevention of deadlocks in packet-switched data transport systems", *IEEE Trans. on Communications*, COM-29:4 (Abril), 512-524.
Hagersten, E., e M. Koster [1998]. "WildFire: A scalable path for SMPs", *Proc. Fifth Int'l. Symposium on High-Performance Computer Architecture*, 9-12 de Janeiro de 1999, Orlando, Fla.
Hagersten, E., A. Landin, e S. Haridi [1992]. "DDM—a cache-only memory architec-ture", *IEEE Computer*, 25:9 (Setembro), 44-54.
Hamacher, V. C., Z. G. Vranesic, e S. G. Zaky [1984]. *Computer Organization*, 2nd ed., New York, McGraw-Hill.
Hamilton, J. [2009]. "Data center networks are in my way", paper presented at the Stanford Clean Slate CTO Summit, Outubro 23, 2009 (*http://mvdirona.com/jrh/TalksAndPapers/JamesHamilton_CleanSlateCTO2009.pdf*).
Hamilton, J. [2010]. "Cloud computing economies of scale", paper presented at the *AWS Workshop on Genomics and Cloud Computing*, 8 de Junho de 2010, Seattle, Wash (http://mvdirona.com/jrh/TalksAndPapers/James-Hamilton_GenomicsCloud20100608.pdf).
Handy, J. [1993]. *The Cache Memory Book*, Academic Press, Boston.
Hauck, E. A., e B. A. Dent [1968]. "Burroughs' B6500/B7500 stack mechanism", *Proc. AFIPS Spring Joint Computer Conf.*, 30 de Abril - 2 de Maio de 1968, Atlantic City, N.J., 245-251.
Heald, R., K. Aingaran, C. Amir, M. Ang, M. Boland, A. Das, P. Dixit, G. Gouldsberry, J. Hart, T. Horel, W.-J. Hsu, J. Kaku, C. Kim, S. Kim, F. Klass, H. Kwan, R. Lo, H. McIntyre, A. Mehta, D. Murata, S. Nguyen, Y.-P. Pai, S. Patel, K. Shin, K. Tam, S. Vishwanthaiah, J. Wu, G. Yee, e H. You [2000]. "Implementation of third-generation SPARC V9 64-b microprocessor", *ISSCC Digest of Technical Papers*, 412-413 e suplemento de slides.
Heinrich, J. [1993]. *MIPS R4000 User's Manual*, Prentice Hall, Englewood Cliffs, N.J..
Henly, M., e B. McNutt [1989]. "*DASD I/O Characteristics: A Comparison of MVS to VM*", Tech. Rep. TR 02.1550 (Maio), IBM General Products Division, San Jose, Calif. Hennessy, J. [1984]. "VLSI processor architecture", *IEEE Trans. on Computers*, C-33:11 (Dezembro), 1221-1246.
Hennessy, J. [1985]. "VLSI RISC processors", *VLSI Systems Design*, 6:10 (Outubro), 22-32.
Hennessy, J., N. Jouppi, F. Baskett, e J. Gill [1981]. "MIPS: A VLSI processor architec-ture", in *CMU Conference on VLSI Systems and Computations*, Computer Science Press, Rockville, Md.
Hewlett-Packard [1994]. *PA-RISC 2.0 Architecture Reference Manual*, 3rd ed., Hewlett-Packard: Palo Alto, Calif.
Hewlett-Packard. [1998]. "HP's '5NINES:5MINUTES' Vision Extends Leadership and Redefines High Availability in Mission-Critical Environments", 1 0 de Fevereiro, *www.future.enterprisecomputing.hp.com/ia64/news/5nines_vision_pr.html*.
Hill, M. D. [1987]. "Aspects of Cache Memory and Instruction Buffer Performance", Tese de doutorado, Tech. Rep. UCB/CSD 87/381, Computer Science Division, University of California, Berkeley.
Hill, M. D. [1988]. "A case for direct mapped caches", *Computer*, 21:12 (Dezembro), 25-40.
Hill, M. D. [1998]. "Multiprocessors should support simple memory consistency models", *IEEE Computer*, 31:8 (Agosto), 28-34.
Hillis, W. D. [1985]. *The Connection Multiprocessor*, Cambridge, Mass: MIT Press.
Hillis, W. D. e G. L. Steele [1986]. "Data parallel algorithms", *Communications of the ACM* 29:12 (Dezembro), 1170-1183. (http://doi.acm.org/10.1145/7902.7903). Hinton, G., D. Sager, M. Upton, D. Boggs, D. Carmean, A. Kyker, e P. Roussel [2001]. "The microarchitecture of the Pentium 4 processor", *Intel Technology Journal*, Fevereiro.
Hintz, R. G., e D. P. Tate [1972]. "Control data STAR-100 processor design", *Proc. IEEE COMPCON*12-14 de Setembro de 1972, San Francisco, 1-4.
Hirata, H., K. Kimura, S. Nagamine, Y. Mochizuki, A. Nishimura, Y. Nakase, e T. Nishizawa [1992]. "An elementary processor architecture with simultaneous instruc-tion issuing from multiple threads", *Proc. 19th Annual Int'l. Symposium on Computer Architecture (ISCA)*19-21 de Maio de 1992, Gold Coast, Australia, 136-145.
Hitachi. [1997]. *SuperH RISC Engine SH7700 Series Programming Manual*, Hitachi, Santa Clara, Calif. (veja *www.halsp.hitachi.com/tech_prod/* e busque pelo título).
Ho, R., K. W. Mai, e M. A. Horowitz [2001]. "The future of wires", *Proc. of the IEEE*, 89:4 (Abril), 490-504.

Hoagland, A. S. [1963]. *Digital Magnetic Recording*, New York: Wiley.
Hockney, R. W., e C. R. Jesshope [1988]. *Parallel Computers 2: Architectures, Pro-gramming and Algorithms*, Adam Hilger, Ltd., Bristol, England.
Holland, J. H. [1959]. "A universal computer capable of executing an arbitrary number of subprograms simultaneously", *Proc. East Joint Computer Conf.* 16, 108-113.
Holt, R. C. [1972]. "Some deadlock properties of computer systems", *ACM Computer Surveys*, 4:3 (Setembro), 179-196.
Hopkins, M. [2000]. "A critical look at IA-64: Massive resources, massive ILP, but can it deliver?" *Microprocessor Report*, Fevereiro.
Hord, R. M. [1982]. *The Illiac-IV, The First Supercomputer*, Computer Science Press, Rockville, Md.
Horel, T., e G. Lauterbach [1999]. "UltraSPARC-III: Designing third-generation 64-bit performance", *IEEE Micro*, 19:3 (Maio-Junho), 73-85.
Hospodor, A. D., e A. S. Hoagland [1993]. "The changing nature of disk controllers", *Proc. IEEE*, 81:4 (Abril), 586-594.
Hölzle, U. [2010]. "Brawny cores still beat wimpy cores, most of the time", *IEEE Micro*, 30:4 (Julho/Agosto).
Hristea, C., D. Lenoski, e J. Keen [1997]. "Measuring memory hierarchy performance of cache-coherent multiprocessors using micro benchmarks", *Proc. ACM/IEEE Conf. on Supercomputing*, 16-21 de Novembro de 1997, San Jose, Calif.
Hsu, P. [1994]. "Designing the TFP microprocessor", *IEEE Micro*, 18:2 (Abril), 2333.
Huck, J., et al. [2000]. "Introducing the IA-64 Architecture", *IEEE Micro*, 20:5 (Setembro-Outubro), 12-23.
Hughes, C. J., P. Kaul, S. V. Adve, R. Jain, C. Park, e J. Srinivasan [2001]. "Variability in the execution of multimedia applications and implications for architecture", *Proc. 28th Annual Int'l. Symposium on Computer Architecture (ISCA)*, 30 de Junho - 4 de Julho de 2001, Goteborg, Suécia, 254-265.
Hwang, K. [1979]. *Computer Arithmetic: Principles, Architecture, and Design*, Wiley, New York.
Hwang, K. [1993]. *Advanced Computer Architecture and Parallel Programming*, McGraw-Hill, New York.
Hwu, W. -M., e Y. Patt [1986]. "HPSm, a high performance restricted data flow archi-tecture having minimum functionality", *Proc. 13th Annual Int'l. Symposium on Com-puter Architecture (ISCA)*, 2-5 de Junho de 1986, Tóquio, 297-307.
Hwu, W. W., S. A. Mahlke, W. Y. Chen, P. P. Chang, N. J. Warter, R. A. Bringmann, R. O. Ouellette, R. E. Hank, T. Kiyohara, G. E. Haab, J. G. Holm, e D. M. Lavery [1993]. "The superblock: An effective technique for VLIW and superscalar compilation", *J. Supercomputing*, 7:1 2 (Março), 229-248.
IBM. [1982]. *The Economic Value of Rapid Response Time*, GE20-0752-0, IBM, White Plains, N.Y., 11-82.
IBM [1990]. The IBM RISC System/6000 processor" (collection of papers), *IBM J. Research and Development*, 34:1 (Janeiro).
IBM [1994]. *The PowerPC Architecture*, Morgan Kaufmann, San Francisco.
IBM [2005]. "Blue Gene", *IBM J. Research and Development* 49:2/3, (edição especial). IEEE [1985]. "IEEE standard for binary floating-point arithmetic", *SIGPLAN Notices* 22:2, 9-25.
IEEE [2005]. "Intel virtualization technology, computer", *IEEE Computer Society*, 38:5 (Maio), 48-56.
IEEE. 754-2008 Working Group. [2006]. "DRAFT Standard for Floating-Point Arithme-tic 754-2008", *http://dx.doi.org/10.1109/IEEESTD.2008.4610935*.
Imprimis Product Specification, 97209 Sabre Disk Drive IPI-2 Interface 1.2 GB, Document No. 64402302, Imprimis, Dallas, Tex.
InfiniBand Trade Association. [2001]. *InfiniBand Architecture Specifications Release 1.0.a*, *www.infinibandta.org*.
Intel. [2001]. "Using MMX Instructions to Convert RGB to YUV Color Conversion", *cedar.intel.com/cgi-bin/ids.dll/content/content.jsp?cntKey=Legacy::irtm_AP548_9996&cntType=IDS_EDITORIAL*.
Internet Retailer. [2005]. "The Gap launches a new site—after two weeks of downtime", Internet® Retailer, Setembro 28, *http://www.internetretailer.com/2005/09/28/the-gap-launches-a-new-site-after-two-weeks-of-downtime*.
Jain, R. [1991]. *The Art of Computer Systems Performance Analysis: Techniques for Experimental Design, Measurement, Simulation, and Modeling*, Wiley, New York.
Networks on Chips. A. Jantsch, e H. Tenhunen (eds.) [2003]. Kluwer Academic Publishers, The Netherlands.
Jimenez, D. A., e C. Lin [2002]. "Neural methods for dynamic branch prediction", *ACM Trans. on Computer Systems*, 20:4 (Novembro), 369-397.
Johnson, M. [1990]. *Superscalar Microprocessor Design*, Prentice Hall, Englewood Cliffs, N.J.
Jordan, H. F. [1983]. "Performance measurements on HEP—a pipelined MIMD computer", *Proc. 10th Annual Int'l. Symposium on Computer Architecture (ISCA)*, 5-7 de Junho de 1982, Estocolmo, Suécia, 207-212.
Jordan, K. E. [1987]. "Performance comparison of large-scale scientific processors: Scalar mainframes, mainframes with vector facilities, and supercomputers", *Computer*, 20:3 (Março), 10-23.
Jouppi, N. P. [1990]. "Improving direct-mapped cache performance by the addition of a small fully-associative cache and prefetch buffers", *Proc. 17th Annual Int'l. Sympo-sium on Computer Architecture (ISCA)*, 28-31 de Maio de 1990, Seattle, Wash., 364-373.
Jouppi, N. P. [1998]. "Retrospective: Improving direct-mapped cache performance by the addition of a small fully-associative cache and prefetch buffers", *25 Years of the Inter-national Symposia on Computer Architecture (Selected Papers)*, ACM, New York, 71-73.

Jouppi, N. P., e D. W. Wall [1989]. "Available instruction-level parallelism for super-scalar and superpipelined processors", *Proc. Third Int'l. Conf. on Architectural Sup-port for Programming Languages and Operating Systems (ASPLOS)*, 3-6 de Abril de 1989, Boston, 272-282.

Jouppi, N. P., e S. J. E. Wilton [1994]. "Trade-offs in two-level on-chip caching", *Proc. 21st Annual Int'l. Symposium on Computer Architecture (ISCA)*, 18-21 de Abril de 1994, Chicago, 34-45.

Kaeli, D. R., e P. G. Emma [1991]. "Branch history table prediction of moving target branches due to subroutine returns", *Proc. 18th Annual Int'l. Symposium on Computer Architecture (ISCA)*, 27-30 de Maio de 1991, Toronto, Canadá, 34-42.

Kahan, J. [1990]. "On the advantage of the 8087's stack", unpublished course notes, Com-puter Science Division, University of California, Berkeley.

Kahan, W. [1968]. "7094-II system support for numerical analysis", *SHARE Secretarial Distribution* SSD-159, Department of Computer Science, University of Toronto.

Kahaner, D. K. [1988]. "Benchmarks for 'real' programs", *SIAM News*, Novembro.

Kahn, R. E. [1972]. "Resource-sharing computer communication networks", *Proc. IEEE*, 60:11 (Novembro), 1397-1407.

Kane, G. [1986]. *MIPS R2000 RISC Architecture*, Prentice Hall, Englewood Cliffs, N.J. Kane, G. [1996]. *PA-RISC 2.0 Architecture*, Prentice Hall, Upper Saddle River, N.J. Kane, G., e J. Heinrich [1992]. *MIPS RISC Architecture*, Prentice Hall, Englewood Cliffs

Katz, R. H., D. A. Patterson, e G. A. Gibson [1989]. "Disk system architectures for high performance computing", *Proc. IEEE*, 77:12 (Dezembro), 1842-1858.

Keckler, S. W., e W. J. Dally [1992]. "Processor coupling: Integrating compile time and runtime scheduling for parallelism", *Proc. 19th Annual Int'l. Symposium on Com-puter Architecture (ISCA)*, 19-21 de Maio de 1992, Gold Coast, Australia, 202-213.

Keller, R. M. [1975]. "Look-ahead processors", *ACM Computing Surveys*, 7:4 (Dezembro), 177-195.

Keltcher, C. N., K. J. McGrath, A. Ahmed, e P. Conway [2003]. "The AMD Opteron processor for multiprocessor servers", *IEEE Micro*, 23:2 (Março-Abril), 66-76dx.doi.org/10.1109.MM.2003.119116.

Kembel, R. [2000]. "Fibre Channel: A comprehensive introduction", *Internet Week*, Abril.

Kermani, P., e L. Kleinrock [1979]. "Virtual Cut-Through: A New Computer Communication Switching Technique", *Computer Networks*, 3:(Janeiro), 267-286.

Kessler, R. [1999]. "The Alpha 21264 microprocessor", *IEEE Micro*, 19:2 (Março/Abril), 24-36.

Kilburn, T., D. B. G. Edwards, M. J. Lanigan, e F. H. Sumner [1962]. "One-level stor-age system", *IRE Trans. on Electronic Computers* EC-11 (Abril) 223-235. Também aparece em D. P. Siewiorek, C. G. Bell, e A. Newell, *Computer Structures: Princi-ples and Examples*, McGraw-Hill, New York, 1982, 135-148.

Killian, E. [1991]. "MIPS R4000 technical overview-64 bits/100 MHz or bust", *Hot Chips III Symposium Record*, Agosto 26-27, 1991, Stanford University, Palo Alto, Calif., 1.6-1.19.

Kim, M. Y. [1986]. "Synchronized disk interleaving", *IEEE Trans. on Computers*, C-35:11 (Novembro), 978-988.

Kissell, K. D. [1997]. "MIPS16: High-density for the embedded market", *Proc. Real Time Systems, 97*, 15 de Junho de 1997, Las Vegas, Nev. (veja *www.sgi.com/MIPS/arch/MIPS16/MIPS16.whitepaper.pdf*).

Kitagawa, K., S. Tagaya, Y. Hagihara, e Y. Kanoh [2003]. A hardware overview of SX-6 and SX-7 supercomputer", *NEC Research & Development J*, 44:1 (Janeiro), 2-7.

Knuth, D. [1981], *The Art of Computer Programming* (Vol. II), 2nd ed., Addison-Wesley: Reading, Mass.

Kogge, P. M. [1981]. *The Architecture of Pipelined Computers*, McGraw-Hill, New York. Kohn, L., e S. -W. Fu [1989]. "A 1,000,000 transistor microprocessor", *Proc. of IEEE Int'l. Symposium on Solid State Circuits (ISSCC)*,15-17 de Fevereiro de 1989, New York, 54-55.

Kohn, L., e N. Margulis [1989]. "Introducing the Intel i860 64-Bit Microprocessor", *IEEE Micro*, 9:4 (Julho), 15-30.

Kontothanassis, L., G. Hunt, R. Stets, N. Hardavellas, M. Cierniak, S. Parthasarathy, W. Meira, S. Dwarkadas, e M. Scott [1997]. "VM-based shared memory on low-latency, remote-memory-access networks", *Proc. 24th Annual Int'l. Symposium on Computer Architecture (ISCA)*, 2-4 de Junho de 1997, Denver, Colo.

Koren, I. [1989]. *Computer Arithmetic Algorithms*, Prentice Hall, Englewood Cliffs, N.J.

Kozyrakis, C. (2000). "Vector IRAM: A media-oriented vector processor with embedded DRAM", paper presented at Hot Chips, 12, 13-15 de Agosto de 2000, Palo Alto, Calif, 13-15.

Kozyrakis, C., e D. Patterson [2002]. "Vector vs. superscalar and VLIW architectures for embedded multimedia benchmarks", *Proc. 35th Annual Int'l. Symposium on Microarchitecture (MICRO-35)*,18-22 de Novembro de 2002, Istanbul, Turquia, .

Kroft, D. [1981]. "Lockup-free instruction fetch/prefetch cache organization", *Proc. Eighth Annual Int'l. Symposium on Computer Architecture (ISCA)*, 12-14 de Maio de 1981, Minneapolis, Minn, 81-87.

Kroft, D. (1998). "Retrospective: Lockup-free instruction fetch/prefetch cache organization", *25 Years of the International Symposia on Computer Architecture (Selected Papers)*, ACM, New York, 20-21.

Kuck, D., P. P. Budnik, S. -C. Chen, D. H. Lawrie, R. A. Towle, R. E. Strebendt, E. W. Davis, Jr., J. Han, P. W. Kraska, e Y. Muraoka [1974]. "Measurements of parallel-ism in ordinary FORTRAN programs", *Computer*, 7:1 (Janeiro), 37-46.

Kuhn, D. R. [1997]. "Sources of failure in the public switched telephone network", *IEEE Computer*, 30:4 (Abril), 31-36.

Kumar, A. [1997]. "The HP PA-8000 RISC CPU", *IEEE Micro*, 17:2 (Março/Abril), 27-32.

Kunimatsu, A., N. Ide, T. Sato, Y. Endo, H. Murakami, T. Kamei, M. Hirano, F. Ishihara, H. Tago, M. Oka, A. Ohba, T. Yutaka, T. Okada, e M. Suzuoki [2000]. "Vector unit architecture for emotion synthesis", *IEEE Micro*, 20:2 (Março–Abril), 40-47.

Kunkel, S. R., e J. E. Smith [1986]. "Optimal pipelining in supercomputers", *Proc. 13th Annual Int'l. Symposium on Computer Architecture (ISCA)*, 2-5 de Junho de 1986, Tóquio, 404-414.

Kurose, J. F., e K. W. Ross [2001]. *Computer Networking: A Top-Down Approach Featuring the Internet*, Addison-Wesley, Boston.

Kuskin, J., D. Ofelt, M. Heinrich, J. Heinlein, R. Simoni, K. Gharachorloo, J. Chapin, D. Nakahira, J. Baxter, M. Horowitz, A. Gupta, M. Rosenblum, e J. L. Hennessy [1994]. "The Stanford FLASH multiprocessor", *Proc. 21st Annual Int'l. Symposium on Computer Architecture (ISCA)*, 18-21 de Abril de 1994, Chicago.

Lam, M. [1988]. "Software pipelining: An effective scheduling technique for VLIW pro-cessors", *SIGPLAN Conf. on Programming Language Design and Implementation*, 22-24 de Junho de 1988, Atlanta, Ga, 318-328.

Lam, M. S., e R. P. Wilson [1992]. "Limits of control flow on parallelism", *Proc. 19th Annual Int'l. Symposium on Computer Architecture (ISCA)*, 19-21 de Maio de 1992, Gold Coast, Australia, 46-57.

Lam, M. S., E. E. Rothberg, e M. E. Wolf [1991]. "The cache performance and optimi-zations of blocked algorithms", *Proc. Fourth Int'l. Conf. on Architectural Support for Programming Languages and Operating Systems (ASPLOS)*, 8-11 de Abril de 1991, Santa Clara, Calif. (*SIGPLAN Notices* 264 (Abril), 63-74), .

Lambright, D. [2000]. "Experiences in measuring the reliability of a cache-based storage system", *Proc. of First Workshop on Industrial Experiences with Systems Software (WIESS 2000), Co-Located with the 4th Symposium on Operating Systems Design and Implementation (OSDI)*, 22 de Outubro de 2000, San Diego, Calif.

Lamport, L. [1979]. "How to make a multiprocessor computer that correctly executes multiprocess programs", *IEEE Trans. on Computers*, C-28:9 (Setembro), 241-248.

Lang, W., J. M. Patel, e S. Shankar [2010]. "Wimpy node clusters: What about non-wimpy workloads? *Proc. Sixth International Workshop on Data Management on New Hardware (DaMoN)*, 7 de Junho, Indianapolis, Ind.

Laprie, J. -C. [1985]. "Dependable computing and fault tolerance: Concepts and terminol-ogy", *Proc. 15th Annual Int'l. Symposium on Fault-Tolerant Computing*, 19-21 de Junho de 1985, Ann Arbor, Mich2-11.

Larson, E. R. [1973]. "Findings of fact, conclusions of law, and order for judgment", *File No. 4-67, Civ. 138, Honeywell v. Sperry-Rand and Illinois Scientific Development*, U.S. District Court for the State of Minnesota, Fourth Division (19 de Outubro).

Laudon, J., e D. Lenoski [1997]. "The SGI Origin: A ccNUMA highly scalable server", *Proc. 24th Annual Int'l. Symposium on Computer Architecture (ISCA)*, 2-4 de Junho de Denver, Colo, 241-251.

Laudon, J., A. Gupta, e M. Horowitz [1994]. "Interleaving: A multithreading technique targeting multiprocessors and workstations", *Proc. Sixth Int'l. Conf. on Architectural Support for Programming Languages and Operating Systems (ASPLOS)*, 4-7 de Outubro, San Jose, Calif, 308-318.

Lauterbach, G., e T. Horel [1999]. "UltraSPARC-III: Designing third generation 64-bit performance", *IEEE Micro*, 19:3 (Maio/Junho).

Lazowska, E. D., J. Zahorjan, G. S. Graham, e K. C. Sevcik [1984]. *Quantitative Sys-tem Performance: Computer System Analysis Using Queueing Network Models*, Prentice Hall, Englewood Cliffs, N.J. (Embora esteja esgotado, ele está dispnível online em *www.cs.washington.edu/homes/lazowska/qsp/*).

Lebeck, A. R., e D. A. Wood [1994]. "Cache profiling and the SPEC benchmarks: A case study", *Computer*, 27:10 (Outubro), 15-26.

Lee, R. [1989]. "Precision architecture", *Computer*, 22:1 (Janeiro), 78-91.

Lee, W. V., et al. [2010]. "Debunking the 100X GPU vs. CPU myth: An evaluation of throughput computing on CPU and GPU", *Proc. 37th Annual Int'l. Symposium on Computer Architecture (ISCA)*, 19-23 de Junho de 2010, Saint-Malo, França.

Leighton, F. T. [1992]. *Introduction to Parallel Algorithms and Architectures: Arrays, Trees, Hypercubes*, Morgan Kaufmann, San Francisco.

Leiner, A. L. [1954]. "System specifications for the DYSEAC", *J. ACM*, 1:2 (Abril), 57-81.

Leiner, A. L., e S. N. Alexander [1954]. "System organization of the DYSEAC", *IRE Trans. of Electronic Computers*, EC-3:1 (Março), 1-10.

Leiserson, C. E. [1985]. "Fat trees: Universal networks for hardware-efficient supercom-puting", *IEEE Trans. on Computers*, C-34:10 (Outubro), 892-901.

Lenoski, D., J. Laudon, K. Gharachorloo, A. Gupta, e J. L. Hennessy [1990]. "The Stan-ford DASH multiprocessor", *Proc. 17th Annual Int'l. Symposium on Computer Archi-tecture (ISCA)*, 28-31 de Maio de 1990, Seattle, Wash., 148-159.

Lenoski, D., J. Laudon, K. Gharachorloo, W. -D. Weber, A. Gupta, J. L. Hennessy, M. A. Horowitz, e M. Lam [1992]. "The Stanford DASH multiprocessor", *IEEE Com-puter*, 25:3 (Março), 63-79.

Levy, H., e R. Eckhouse [1989]. *Computer Programming and Architecture: The VAX*, Boston: Digital Press.

Li, K. [1988]. "IVY: A shared virtual memory system for parallel computing", *Proc. 1988 Int'l. Conf. on Parallel Processing*, Pennsylvania State University Press, University Park, Penn.

Li, S., K. Chen, J. B. Brockman, e N. Jouppi (2011). "Performance Impacts of Non-blocking Caches in Out-of-order Processors", HP Labs Tech Report HPL-2011-65 (O texto completo está disponível em http://Library.hp.com/techpubs/2011/Hpl-2011-65.html).

Lim, K., P. Ranganathan, J. Chang, C. Patel, S. Mudge, e S. Reinhardt [2008]. "Under-standing and designing new system architectures for emerging warehouse-computing environments", *Proc. 35th Annual Int'l. Symposium on Computer Architecture (ISCA)*, 21-25 de Junho de 2008, Beijing, China.

Lincoln, N. R. [1982]. "Technology and design trade offs in the creation of a modern supercomputer", *IEEE Trans. on Computers*, C-31:5 (Maio), 363-376.

Lindholm, T., e F. Yellin [1999]. *The Java Virtual Machine Specification*, 2nd ed., Addison-Wesley. Reading, Mass (Também disponível online em *java.sun.com/docs/books/ vmspec/*).

Lipasti, M. H., e J. Shen [1996]. "Exceeding the dataflow limit via value prediction", *Proc. 29th Int'l. Symposium on Microarchitecture*, Dezembro 2-4, 1996, Paris, França.

Lipasti, M. H., C. B. Wilkerson, e J. P. Shen [1996]. "Value locality and load value pre-diction", *Proc. Seventh Conf. on Architectural Support for Programming Languages and Operating Systems (ASPLOS)*, 1-5 de Outubro de 1996, Cambridge, Mass., 138-147.

Liptay, J. S. [1968]. "Structural aspects of the System/360 Model 85, Part II: The cache", *IBM Systems J* 7:1, 15-21.

Lo, J., L. Barroso, S. Eggers, K. Gharachorloo, H. Levy, e S. Parekh [1998]. "An analy-sis of database workload performance on simultaneous multithreaded processors", *Proc. 25th Annual Int'l. Symposium on Computer Architecture (ISCA)*, 3-14 de Julho de 1998, Barcelona, Espanha, 39-50.

Lo, J., S. Eggers, J. Emer, H. Levy, R. Stamm, e D. Tullsen [1997]. "Converting thread-level parallelism into instruction-level parallelism via simultaneous multithreading", *ACM Trans. on Computer Systems*, 15:2 (Agosto), 322-354.

Lovett, T., e S. Thakkar [1988]. "The Symmetry multiprocessor system", *Proc. 1988 Int'l. Conf. of Parallel Processing*, University Park, Penn., 303-310.

Lubeck, O., J. Moore, e R. Mendez [1985]. "A benchmark comparison of three super-computers: Fujitsu VP-200, Hitachi S810/20, e Cray X-MP/2", *Computer*, 18:12 (Dezembro), 10-24.

Luk, C. -K., e T. C. Mowry [1999]. "Automatic compiler-inserted prefetching for pointer-based applications", *IEEE Trans. on Computers*, 48:2 (Fevereiro), 134-141.

Lunde, A. [1977]. "Empirical evaluation of some features of instruction set processor architecture", *Communications of the ACM*, 20:3 (Março), 143-152.

Luszczek, P., J. J. Dongarra, D. Koester, R. Rabenseifner, B. Lucas, J. Kepner, J. McCal-pin, D. Bailey, e D. Takahashi [2005]. "Introduction to the HPC challenge bench-mark suite", Lawrence Berkeley National Laboratory, Paper LBNL-57493 (Abril 25), *repositories.cdlib.org/lbnl/LBNL-57493*.

Maberly, N. C. [1966]. *Mastering Speed Reading*, American Library, New York New.

Magenheimer, D. J., L. Peters, K. W. Pettis, e D. Zuras [1988]. "Integer multiplication and division on the HP precision architecture", *IEEE Trans. on Computers* 37:8, 980-990.

Mahlke, S. A., W. Y. Chen, W. -M. Hwu, B. R. Rau, e M. S. Schlansker [1992]. "Sentinel scheduling for VLIW and superscalar processors", *Proc. Fifth Int'l. Conf. on Architec-tural Support for Programming Languages and Operating Systems (ASPLOS)*, 12-15 de Outubro de 1992, Boston, 238-247.

Mahlke, S. A., R. E. Hank, J. E. McCormick, D. I. Agosto, e W. W. Hwu [1995]. "A comparison of full and partial predicated execution support for ILP processors", *Proc. 22nd Annual Int'l. Symposium on Computer Architecture (ISCA)*, 22-2 4 de Junho de 1995, Santa Margherita, Itália, 138-149.

Major, J. B. [1989]. "Are queuing models within the grasp of the unwashed? *Proc. Int'l. Conf. on Management and Performance Evaluation of Computer Systems*, 11-15 de Dezembro de 1989, Reno, Nev., 831-839.

Markstein, P. W. [1990]. "Computation of elementary functions on the IBM RISC Sys-tem/6000 processor", *IBM J. Research and Development* 34:1, 111-119.

Mathis, H. M., A. E. Mercias, J. D. McCalpin, R. J. Eickemeyer, e S. R. Kunkel [2005]. Characterization of the multithreading (SMT) efficiency in Power5", *IBM J. Research and Development*, 49:4/5 (Julho/Setembro), 555-564.

McCalpin, J. [2005]. "STREAM: Sustainable Memory Bandwidth in High Performance Computers", www.cs.virginia.edu/stream/.

McCalpin, J., D. Bailey, e D. Takahashi [2005]. *Introduction to the HPC Challenge Benchmark Suite*, Paper LBNL-57493 Lawrence Berkeley National Laboratory, University of California, Berkeley, *repositories.cdlib.org/lbnl/LBNL-57493.*

McCormick, J., e A. Knies [2002]. "A brief analysis of the SPEC CPU2000 benchmarks on the Intel Itanium 2 processor", paper presented at Hot Chips 14, 18 - 20 de Agosto de 2002, Stanford University, Palo Alto, Calif.

McFarling, S. [1989]. "Program optimization for instruction caches", *Proc. Third Int'l. Conf. on Architectural Support for Programming Languages e Operating Systems (ASPLOS)*, 3-6 de Abril de 1989, Boston, 183-191.

McFarling, S. [1993]. *Combining Branch Predictors*, WRL Technical Note TN-36, Digital Western Research Laboratory, Palo Alto, Calif.

McFarling, S., e J. Hennessy [1986]. "Reducing the cost of branches", *Proc. 13th Annual Int'l. Symposium on Computer Architecture (ISCA)*, 2-5 de Junho de 1986, Tóquio, 396-403.

McGhan, H., e M. O'Connor [1998]. "PicoJava: A direct execution engine for Java bytecode", *Computer*, 31:10 (Outubro), 22-30.

McKeeman, W. M. [1967]. "Language directed computer design", *Proc. AFIPS Fall Joint Computer Conf.*, 14-16 de Novembro de 1967, Washington, D.C., 413-417.

McMahon, F. M. [1986]. "*The Livermore FORTRAN Kernels: A Computer Test of Numer-ical Performance Range*", Tech. Rep. UCRL-55745, Lawrence Livermore National Laboratory, University of California, Livermore.

McNairy, C., e D. Soltis [2003]. "Itanium 2 processor microarchitecture", *IEEE Micro*, 23:2 (Março-Abril), 44-55.
Mead, C., e L. Conway [1980]. *Introduction to VLSI Systems*, Addison-Wesley, Reading, Mass.
Mellor-Crummey, J. M., e M. L. Scott [1991]. "Algorithms for scalable synchronization on shared-memory multiprocessors", *ACM Trans. on Computer Systems*, 9:1 (Fevereiro), 21-65.
Menabrea, L. F. [1842]. "Sketch of the analytical engine invented by Charles Babbage", *Bibliothèque Universelle de Genève*, 82:Outubro.
Menon, A., J. Renato Santos, Y. Turner, G. Janakiraman, e W. Zwaenepoel [2005]. "Diagnosing performance overheads in the xen virtual machine environment", *Proc. First ACM/USENIX Int'l. Conf. on Virtual Execution Environments*, 11-12 de Junho de 2005, Chicago, 13-23.
Merlin, P. M., e P. J. Schweitzer [1980]. "Deadlock avoidance in store-and-forward net-works. Part I. Store-and-forward deadlock", *IEEE Trans. on Communications*, COM-28:3 (Março), 345-354.
Metcalfe, R. M. [1993]. "Computer/network interface design: Lessons from Arpanet and Ethernet", *IEEE J. on Selected Areas in Communications*, 11:2 (Fevereiro), 173-180.
Metcalfe, R. M., e D. R. Boggs [1976]. "Ethernet: Distributed packet switching for local computer networks", *Communications of the ACM*, 19:7 (Julho), 395-404.
Metropolis N., J. Howlett, e G. C. Rota (eds.) [1980]. *A History of Computing in the Twentieth Century*, Academic Press, New York.
Meyer, R. A., e L. H. Seawright [1970]. A virtual machine time sharing system, *IBM Systems J* 9:3, 199-218.
Meyers, G. J. [1978]. "The evaluation of expressions in a storage-to-storage architecture", *Computer Architecture News*, 7:3 (Outubro), 20-23.
Meyers, G. J. [1982]. *Advances in Computer Architecture*, 2nd ed., Wiley, New York.
Micron. (2004). "Calculating Memory System Power for DDR2", *http://download.micron.com/pdf/pubs/designline/dl1Q04.pdf*.
Micron. (2006). "The Micron® System-Power Calculator", *http://www.micron.com/ systemcalc*.
MIPS. (1997). "MIPS16 Application Specific Extension Product Description", *www.sgi.com/MIPS/arch/MIPS16/mips16.pdf*.
Miranker, G. S., J. Rubenstein, e J. Sanguinetti [1988]. "Squeezing a Cray-class super-computer into a single-user package", *Proc. IEEE COMPCON*, 29 de Fevereiro - 4 de Março de 1988, San Francisco, 452-456.
Mitchell, D. [1989]. "The Transputer: The time is now", *Computer Design* (RISC suppl), 40-41.
Mitsubishi [1996]. *Mitsubishi 32-Bit Single Chip Microcomputer M32R Family Software Manual*, Mitsubishi, Cypress, Calif.
Miura, K., K. Uchida [1983]. "FACOM vector processing system: VP100/200", *Proc. NATO Advanced Research Workshop on High-Speed Computing, 20 - 22 de Junho de 1983, Jülich, Alemanha Ocidental*. Também aparece em K. Hwang, ed., "Superprocessors: Design and applications", *IEEE* (Agosto 1984), 59-73.
Miya, E. N. [1985]. "Multiprocessor/distributed processing bibliography", *Computer Architecture News* 13:1, 27-29.
Montoye, R. K., E. Hokenek, e S. L. Runyon [1990]. "Design of the IBM RISC System/ 6000 floating-point execution", *IBM J. Research and Development* 34:1, 59-70.
Moore, B., A. Padegs, R. Smith, e W. Bucholz [1987]. "Concepts of the System/370 vector architecture", *14th Annual Int'l. Symposium on Computer Architecture (ISCA)*, 2-5 de Junho de 1987, Pittsburgh, Penn., 282-292.
Moore, G. E. [1965]. "Cramming more components onto integrated circuits", *Electronics*, 38:8 (Abril 19), 114-117.
Morse, S., B. Ravenal, S. Mazor, e W. Pohlman [1980]. "Intel microprocessors—8080 to 8086", *Computer* 13:10 (Outubro).
Moshovos, A., e G. S. Sohi [1997]. "Streamlining inter-operation memory communica-tion via data dependence prediction", *Proc. 30th Annual Int'l. Symposium on Micro-architecture*, Dezembro 1-3, Research Triangle Park, N.C., 235-245.
Moshovos, A., S. Breach, T. N. Vijaykumar, e G. S. Sohi [1997]. "Dynamic speculation and synchronization of data dependences", *24th Annual Int'l. Symposium on Com-puter Architecture (ISCA)*, 2-4 de Junho de 1997, Denver, Colo.
Moussouris, J., L. Crudele, D. Freitas, C. Hansen, E. Hudson, S. Przybylski, T. Riordan, e C. Rowen [1986]. "A CMOS RISC processor with integrated system functions", *Proc. IEEE COMPCON*, Março 3-6, 1986, San Francisco, 191.
Mowry, T. C., S. Lam, e A. Gupta [1992]. "Design and evaluation of a compiler algo-rithm for prefetching", *Proc. Fifth Int'l. Conf. on Architectural Support for Program-ming Languages and Operating Systems (ASPLOS)*, 12-15 de Outubro de 1992, Boston (SIGPLAN Notices 279 (Setembro), 62-73).
MSN Money. (2005). "Amazon Shares Tumble after Rally Fizzles", *http://moneycentral.msn.com/content/CNBCTV/Articles/Dispatches/P133695.asp*.
Muchnick, S. S. [1988]. "Optimizing compilers for SPARC", *Sun Technology*, 1:3 (Summer), 64-77.
Mueller, M., L. C. Alves, W. Fischer, M. L. Fair, e I. Modi [1999]. "RAS strategy for IBM S/390 G5 and G6", *IBM J. Research and Development*, 43:5-6 (Setembro-Novembro), 875-888.
Mukherjee, S. S., C. Weaver, J. S. Emer, S. K. Reinhardt, e T. M. Austin [2003]. "Mea-suring architectural vulnerability factors", *IEEE Micro* 23:6, 70-75.
Murphy, B., e T. Gent [1995]. "Measuring system and software reliability using an automated data collection process", *Quality and Reliability Engineering International*, 11:5 (Setembro-Outubro), 341-353.

Myer, T. H., e I. E. Sutherland [1968]. "On the design of display processors", *Communications of the ACM*, 11:6 (Junho), 410-414.

Narayanan, D., E. Thereska, A. Donnelly, S. Elnikety, e A. Rowstron [2009]. "Migrating server storage to SSDs: Analysis of trade-offs", *Proc. 4th ACM European Conf. on Computer Systems*, 1-3 de Abril de 2009, Nuremberg, Alemanha.

National Research Council. [1997]. *The Evolution of Untethered Communications*, Computer Science and Telecommunications Board, National Academy Press, Washington, D.C.

National Storage Industry Consortium. (1998). "Tape Roadmap", *www.nsic.org*.

Nelson, V. P. [1990]. "Fault-tolerant computing: Fundamental concepts", *Computer*, 23:7 (Julho), 19-25.

Ngai, T. -F., e M. J. Irwin [1985]. "Regular, area-time efficient carry-lookahead adders", *Proc. Seventh IEEE Symposium on Computer Arithmetic*, 4-6 de Junho de 1985, University of Illinois, Urbana, 9-15.

Nicolau, A., e J. A. Fisher [1984]. "Measuring the parallelism available for very long instruction word architectures", *IEEE Trans. on Computers*, C-33:11 (Novembro), 968-976.

Nikhil, R. S., G. M. Papadopoulos, e Arvind [1992]. "*T: A multithreaded massively parallel architecture", *Proc. 19th Annual Int'l. Symposium on Computer Architecture (ISCA)*, 19-21 de Maio de 1992, Gold Coast, Australia, 156-167.

Noordergraaf, L., e R. van der Pas [1999]. "Performance experiences on Sun's WildFire prototype", *Proc. ACM/IEEE Conf. on Supercomputing*, 13-19 de Novembro de 1999, Portland, Ore.

Nyberg, C. R., T. Barclay, Z. Cvetanovic, J. Gray, e D. Lomet [1994]. "AlphaSort: A RISC machine sort", *Proc. ACM SIGMOD*, 24-27 de Maio de 1994, Minneapolis, Minn.

Oka, M., e M. Suzuoki [1999]. "Designing and programming the emotion engine", *IEEE Micro* 19:6 (Novembro-Dezembro), 20-28.

Okada, S., S. Okada, Y. Matsuda, T. Yamada, e A. Kobayashi [1999]. "System on a chip for digital still camera", *IEEE Trans. on Consumer Electronics*, 45:3 (Agosto), 584-590.

Oliker, L., A. Canning, J. Carter, J. Shalf, e S. Ethier [2004]. "Scientific computations on modern parallel vector systems", *Proc. ACM/IEEE Conf. on Supercomputing*, 6-12 de Novembro de 2004, Pittsburgh, Penn., 10.

Pabst, T., [2000]. "Performance Showdown at 133 MHz FSB—The Best Platform for Coppermine", *www6.tomshardware.com/mainboard/00q1/000302/*.

Padua, D., e M. Wolfe [1986]. "Advanced compiler optimizations for supercomputers", *Communications of the ACM*, 29:12 (Dezembro), 1184-1201.

Palacharla, S., e R. E. Kessler [1994]. "Evaluating stream buffers as a secondary cache replacement", *Proc. 21st Annual Int'l. Symposium on Computer Architecture (ISCA)*, 18-21 de Abril de 1994, Chicago, 24-33.

Palmer, J., e S. Morse [1984]. *The 8087 Primer*, John Wiley & Sons, New York, 93.

Pan, S. -T., K. So, e J. T. Rameh [1992]. "Improving the accuracy of dynamic branch prediction using branch correlation", *Proc. Fifth Int'l. Conf. on Architectural Support for Programming Languages e Operating Systems (ASPLOS)*, 12-15 de Outubro de 1992, Boston, 76-84.

Partridge, C. [1994]. *Gigabit Networking*, Addison-Wesley, Reading, Mass.

Patterson, D. [1985]. "Reduced instruction set computers", *Communications of the ACM*, 28:1 (Janeiro), 8-21.

Patterson, D. [2004]. "Latency lags bandwidth", *Communications of the ACM* 47:10 (Outubro), 71-75.

Patterson, D. A., e D. R. Ditzel [1980]. "The case for the reduced instruction set computer", *Computer Architecture News*, 8:6 (Outubro), 25-33.

Patterson, D. A., e J. L. Hennessy [2004]. *Computer Organization and Design: The Hardware/Software Interface*, 3rd ed., Morgan Kaufmann, San Francisco.

Patterson, D. A., G. A. Gibson, e R. H. Katz [1987]. *A Case for Redundant Arrays of Inexpensive Disks (RAID)*, Tech. Rep. UCB/CSD 87/391, University of California, Berkeley. Também aparece em *Proc. ACM SIGMOD*, 1-3 de Junho de 1988, Chicago, 109-116.

Patterson, D. A., P. Garrison, M. Hill, D. Lioupis, C. Nyberg, T. Sippel, e K. Van Dyke [1983]. "Architecture of a VLSI instruction cache for a RISC", *10th Annual Int'l. Conf. on Computer Architecture Conf. Proc.*, 13-16 de Junho de 1983, Estocolmo, Suécia, 108-116.

Pavan, P., R. Bez, P. Olivo, e E. Zanoni [1997]. "Flash memory cells—an overview." *Proc. IEEE*, 85:8 (Agosto), 1248-1271.

Peh, L. S., e W. J. Dally [2001]. "A delay model and speculative architecture for pipe-lined routers", *Proc. 7th Int'l. Symposium on High-Performance Computer Architecture*, 22-24 de Janeiro de 2001, Monterrey, Mexico.

Peng, V., S. Samudrala, e M. Gavrielov [1987]. "On the implementation of shifters, multipliers, and dividers in VLSI floating point units", *Proc. 8th IEEE Symposium on Computer Arithmetic*, 19-21 de Maio de 1987, Como, Itália, 95-102.

Pfister, G. F. [1998]. *In Search of Clusters*, 2nd ed., Prentice Hall, Upper Saddle River, N.J.

Pfister, G. F., W. C. Brantley, D. A. George, S. L. Harvey, W. J. Kleinfekder, K. P. McAuliffe, E. A. Melton, V. A. Norton, e J. Weiss [1985]. "The IBM research parallel processor prototype (RP3): Introduction and architecture", *Proc. 12th Annual Int'l. Symposium on Computer Architecture (ISCA)*, 17-19 de Junho de 1985, Boston, Mass, 764-771.

Pinheiro, E., W. D. Weber, e L. A. Barroso [2007]. "Failure trends in a large disk drive population", *Proc. 5th USENIX Conference on File and Storage Technologies (FAST'07)*, 13-16 de Fevereiro de 2007, San Jose, Calif.

Pinkston, T. M. [2004]. "Deadlock characterization and resolution in interconnection networks", in M. C. Zhu, e M. P. Fanti, eds., *Deadlock Resolution in Computer-Integrated Systems*. CRC Press, Boca Raton, FL, 445-492.

Pinkston, T. M., e J. Shin [2005]. "Trends toward on-chip networked microsystems", *Int'l. J. of High Performance Computing and Networking* 3:1, 3-18.

Pinkston, T. M., e S. Warnakulasuriya [1997]. "On deadlocks in interconnection networks", *24th Annual Int'l. Symposium on Computer Architecture (ISCA)*, 2-4 de Junho de 1997, Denver, Colo.

Pinkston, T. M., A. Benner, M. Krause, I. Robinson, e T. Sterling [2003]. "InfiniBand: The 'de facto' future standard for system and local area networks or just a scalable replacement for PCI buses?", *Cluster Computing,* (edição especial sobre arquitetura de comunicação para clusters) 6:2 Abril, 95-104.

Postiff, M. A., D. A. Greene, G. S. Tyson, e T. N. Mudge [1999]. "The limits of instruc-tion level parallelism in SPEC95 applications", *Computer Architecture News*, 27:1 (Março), 31-40.

Przybylski, S. A. [1990]. *Cache Design: A Performance-Directed Approach*, Morgan Kaufmann, San Francisco.

Przybylski, S. A., M. Horowitz, e J. L. Hennessy [1988]. "Performance trade-offs in cache design", *15th Annual Int'l. Symposium on Computer Architecture*, 30 de Maio - 2 de Junho de 1988, Honolulu, Hawaii, 290-298.

Puente, V., R. Beivide, J. A. Gregorio, J. M. Prellezo, J. Duato, e C. Izu [1999]. "Adap-tive bubble router: A design to improve performance in torus networks", *Proc. 28th Int'l. Conference on Parallel Processing*, 21-24 de Setembro de 1999, Aizu-Wakamatsu, Fukushima, Japão.

Radin, G. [1982]. "The 801 minicomputer", *Proc. Symposium Architectural Support for Programming Languages and Operating Systems (ASPLOS)*, 1-3 de Março de 1982, Palo Alto, Calif., 39-47.

Rajesh Bordawekar, Uday Bondhugula, Ravi Rao: Believe it or not!: mult-core CPUs can match GPU performance for a FLOP-intensive application! 19th International Conference on Parallel Architecture and Compilation Techniques (PACT 2010), Vienna, Austria, 11—15 de Setembro de 2010: 537-538.

Ramamoorthy, C. V., e H. F. Li [1977]. "Pipeline architecture", *ACM Computing Surveys* 9:1 (Março), 61-102.

Ranganathan, P., P. Leech, D. Irwin, e J. Chase [2006]. "Ensemble-Level Power Management for Dense Blade Servers", *Proc. 33rd Annual Int'l. Symposium on Computer Architecture (ISCA)*, 17-21 de Junho de 2006, Boston, Mass., 66-77.

Rau, B. R. [1994]. "Iterative modulo scheduling: An algorithm for software pipelining loops", *Proc. 27th Annual Int'l. Symposium on Microarchitecture*, 30 de Novembro - 2 de Dezembro de 1994, San Jose, Calif, 63-74.

Rau, B. R., C. D. Glaeser, e R. L. Picard [1982]. "Efficient code generation for horizon-tal architectures: Compiler techniques and architectural support", *Proc. Ninth Annual Int'l. Symposium on Computer Architecture (ISCA)*, 26-29 de Abril de 1982, Austin, Tex., 131-139.

Rau, B. R., D. W. L. Yen, W. Yen, e R. A. Towle [1989]. "The Cydra 5 departmental supercomputer: Design philosophies, decisions, and trade-offs", *IEEE Computers*, 22:1 (Janeiro), 12-34.

Reddi, V. J., B. C. Lee, T. Chilimbi, e K. Vaid [2010]. "Web search using mobile cores: Quantifying and mitigating the price of efficiency", *Proc. 37th Annual Int'l. Symposium on Computer Architecture (ISCA)*, 19-23 de Junho de 2010, Saint-Malo, França.

Redmond, K. C., e T. M. Smith [1980]. *Project Whirlwind—The History of a Pioneer Computer*, Digital Press, Boston.

Reinhardt, S. K., J. R. Larus, e D. A. Wood [1994]. "Tempest and Typhoon: User-level shared memory", *21st Annual Int'l. Symposium on Computer Architecture (ISCA)*, 18-21 de Abril de 1994, Chicago, 325-336.

Reinman, G., e N. P. Jouppi. [1999]. "Extensions to CACTI", *research.compaq.com/wrl/people/jouppi/CACTI.html*.

Rettberg, R. D., W. R. Crowther, P. P. Carvey, e R. S. Towlinson [1990]. "The Monarch parallel processor hardware design", *IEEE Computer*, 23:4 (Abril), 18-30.

Riemens, A., K. A. Vissers, R. J. Schutten, F. W. Sijstermans, G. J. Hekstra, e G. D. La Hei [1999]. "Trimedia CPU64 application domain and benchmark suite", *Proc. IEEE Int'l. Conf. on Computer Design: VLSI in Computers and Processors (ICCD'99)*, 10-13 de Outubro de 1999, Austin, Tex., 580-585.

Riseman, E. M., e C. C. Foster [1972]. "Percolation of code to enhance paralled dispatching and execution", *IEEE Trans. on Computers*, C-21:12 (Dezembro), 1411-1415.

Robin, J., e C. Irvine [2000]. "Analysis of the Intel Pentium's ability to support a secure virtual machine monitor", *Proc. USENIX Security Symposium*, 14-17 de Agosto de 2000, Denver, Colo.

Robinson, B., e L. Blount [1986]. *The VM/HPO 3880-23 Performance Results*, IBM Tech. Bulletin GG66-0247-00, IBM Washington Systems Center, Gaithersburg, Md.

Ropers, A., H.W. Lollman, e J. Wellhausen [1999]. *DSPstone: Texas Instruments TMS320C54x*, Tech. Rep. IB 315 1999/9-ISS-Version 0.9, Aachen University of Technology, Aaachen, Alemanha (*www.ert.rwth-aachen.de/Projekte/Tools/coal/ dspstone_c54x/index.html*).

Rosenblum, M., S. A. Herrod, E. Witchel, e A. Gupta [1995]. "Complete computer sim-ulation: The SimOS approach", *IEEE Parallel and Distributed Technology (now called Concurrency)* 4:3, 34-43.

Rowen, C., M. Johnson, e P. Ries [1988]. "The MIPS R3010 floating-point coproces-sor", *IEEE Micro*, 8:3 (Junho), 53-62.

Russell, R. M. [1978]. "The Cray-1 processor system", *Communications of the ACM*, 21:1 (Janeiro), 63-72.

Rymarczyk, J. [1982]. "Coding guidelines for pipelined processors", *Proc. Symposium Architectural Support for Programming Languages and Operating Systems (ASPLOS)*, 1-3 de Março de 1982, Palo Alto, Calif., 12-19.

Saavedra-Barrera, R.H. [1992]. "CPU Performance Evaluation and Execution Time Pre-diction Using Narrow Spectrum Benchmarking", Ph.D. dissertation, University of California, Berkeley.

Salem, K., e H. Garcia-Molina [1986]. "Disk striping", *Proc. 2nd Int'l. IEEE Conf. on Data Engineering*, 5-7 de Fevereiro de 1986, Washington, D.C., 249-259.

Saltzer, J. H., D. P. Reed, e D. D. Clark [1984]. "End-to-end arguments in system design", *ACM Trans. on Computer Systems*, 2:4 (Novembro), 277-288.

Samples, A. D., e P. N. Hilfinger [1988]. *Code Reorganization for Instruction Caches*, Tech. Rep. UCB/CSD 88/447, University of California, Berkeley.

Santoro, M. R., G. Bewick, e M. A. Horowitz [1989]. "Rounding algorithms for IEEE multipliers", *Proc. Ninth IEEE Symposium on Computer Arithmetic*, 6-8 de Setembro, Santa Monica, Calif., 176-183.

Satran, J., D. Smith, K. Meth, C. Sapuntzakis, M. Wakeley, P. Von Stamwitz, R. Haagens, E. Zeidner, L. Dalle Ore, e Y. Klein [2001]. "iSCSI", IPS Working Group of IETF, Internet draft www.ietf.org/internet-drafts/draft-ietf-ips-iscsi-07.txt.

Saulsbury, A., T. Wilkinson, J. Carter, e A. Landin [1995]. "An argument for Simple COMA", *Proc. First IEEE Symposium on High-Performance Computer Architec-tures*, 22-25 de Janeiro de 1995, Raleigh, N.C., 276-285.

Schneck, P. B. [1987]. *Superprocessor Architecture*, Kluwer Academic Publishers, Norwell, Mass.

Schroeder, B., e G. A. Gibson [2007]. "Understanding failures in petascale computers", *J. of Physics Conf. Series* 78:1, 188-198.

Schroeder, B., E. Pinheiro, e W. -D. Weber [2009]. "DRAM errors in the wild: a large-scale field study", *Proc. Eleventh Int'l. Joint Conf. on Measurement and Modeling of Computer Systems (SIGMETRICS)*, 15-19 de Junho de 2009, Seattle, Wash.

Schurman, E., e J. Brutlag [2009]. "The user and business impact of server delays", *Proc. Velocity: Web Performance and Operations Conf*, 22-24 de Junho de 2009, San Jose, Calif.

Schwartz, J. T. [1980]. "Ultracomputers", *ACM Trans. on Programming Languages and Systems* 4:2, 484-521.

Scott, N. R. [1985]. *Computer Number Systems and Arithmetic*, Prentice Hall, Englewood Cliffs, N.J.

Scott, S. L. [1996]. "Synchronization and communication in the T3E multiprocessor", *Seventh Int'l. Conf. on Architectural Support for Programming Languages and Operating Systems (ASPLOS)*, 1—5 de Outubro de 1996, Cambridge, Mass.

Scott, S. L., e J. Goodman [1994]. "The impact of pipelined channels on *k*-ary *n*-cube networks", *IEEE Trans. on Parallel and Distributed Systems*, 5:1 (Janeiro), 1-16.

Scott, S. L., e G. M. Thorson [1996]. "The Cray T3E network: Adaptive routing in a high performance 3D torus", *Proc. IEEE HOT Interconnects*, 9615-17 de Agosto de 1996, Stanford University, Palo Alto, Calif., 14-156.

Scranton, R.A., D.A. Thompson, e D. W. Hunter [1983]. *The Access Time Myth*", Tech. Rep. RC 10197 (45223), IBM, Yorktown Heights, N.Y.

Seagate. [2000]. *Seagate Cheetah 73 Family: ST173404LW/LWV/LC/LCV Product Manual*, Vol. 1, Seagate, Scotts Valley, Calif. (*www.seagate.com/support/disc/manuals/scsi/29478b.pdf*).

Seitz, C. L. [1985]. "The Cosmic Cube (concurrent computing)", *Communications of the ACM*, 28:1 (Janeiro), 22-33.

Senior, J. M. [1993]. *Optical Fiber Commmunications: Principles and Practice*, 2nd ed., Prentice Hall, Hertfordshire, U.K..

Sharangpani, H., e K. Arora [2000]. "Itanium Processor Microarchitecture", *IEEE Micro*, 20:5 (Setembro–Outubro), 24-43.

Shurkin, J. [1984]. *Engines of the Mind: A History of the Computer*, W.W. Norton, New York.

Shustek, L.J. [1978]. "Analysis and Performance of Computer Instruction Sets", Ph.D. dissertation, Stanford University, Palo Alto, Calif.

Silicon Graphics. [1996]. *MIPS V Instruction* Set (veja http://www.sgi.com/MIPS/arch/ISA5/#MIPSV_indx).

Singh, J. P., J. L. Hennessy, e A. Gupta [1993]. "Scaling parallel programs for multipro-cessors: Methodology and examples", *Computer*, 26:7 (Julho), 22-33.

Sinharoy, B., R. N. Koala, J. M. Tendler, R. J. Eickemeyer, e J. B. Joyner [2005]. "POWER5 system microarchitecture", *IBM J. Research and Development* 49:4-5, 505-521.

Sites, R. [1979]. *Instruction Ordering for the CRAY-1 Computer*, Tech. Rep. 78-CS-023, Dept. of Computer Science, University of California, San Diego.

Sites R. L. (ed.) [1992]. *Alpha Architecture Reference Manual.* Digital Press, Burlington, Mass.

Sites R. L., e R. Witek (eds.) [1995]. *Alpha Architecture Reference Manual* 2nd ed., Digital Press, Newton, Mass.

Skadron, K., e D. W. Clark [1997]. "Design issues and tradeoffs for write buffers", *Proc. Third Int'l. Symposium on High-Performance Computer Architecture*, 1-5 de Fevereiro de 1997, San Antonio, Tex., 144-155.

Skadron, K., P. S. Ahuja, M. Martonosi, e D. W. Clark [1999]. "Branch prediction, instruction-window size, and cache size: Performance tradeoffs and simulation techniques", *IEEE Trans. on Computers*, 48:11 (Novembro).

Slater, R. [1987]. *Portraits in Silicon*, MIT Press, Cambridge, Mass.

Slotnick, D. L., W. C. Borck, e R. C. McReynolds [1962]. "The Solomon computer", *Proc. AFIPS Fall Joint Computer Conf*, 4-6 de Dezembro de 1962, Philadelphia, Penn, 97-107.

Smith, A. J. [1982]. "Cache memories", *Computing Surveys*, 14:3 (Setembro), 473-530.

Smith, A., e J. Lee [1984]. "Branch prediction strategies and branch-target buffer design", *Computer*, 17:1 (Janeiro), 6-22.

Smith, B. J. [1978]. "A pipelined, shared resource MIMD computer", *Proc. Int'l. Conf. on Parallel Processing (ICPP)*, Agosto, Bellaire, Mich., 6-8.

Smith, B. J. [1981]. "Architecture and applications of the HEP multiprocessor system", *Real-Time Signal Processing IV*, 298 (Agosto), 241-248.

Smith, J. E. [1981]. "A study of branch prediction strategies", *Proc. Eighth Annual Int'l. Symposium on Computer Architecture (ISCA)*, 12-14 de Maio de 1981, Minneapolis, Minn., 135-148.

Smith, J. E. [1984]. "Decoupled access/execute computer architectures", *ACM Trans. on Computer Systems*, 2:4 (Novembro), 289-308.

Smith, J. E. [1988]. "Characterizing computer performance with a single number", *Communications of the ACM*, 31:10 (Outubro), 1202-1206.

Smith, J. E. [1989]. "Dynamic instruction scheduling and the Astronautics ZS-1", *Computer*, 22:7 (Julho), 21-35.

Smith, J. E., e J. R. Goodman [1983]. "A study of instruction cache organizations and replacement policies", *Proc. 10th Annual Int'l. Symposium on Computer Architecture (ISCA)*, 5-7 de Junho de 1982, Estocolmo, Suécia, 132-137.

Smith, J.E., A. R. Pleszkun [1988]. "Implementing precise interrupts in pipelined processors", *IEEE Trans. on Computers*, 37:5 (Maio), 562-573. (Este artigo é baseado em um artigo anterior que apareceu em *Proc. 12th Annual Int'l. Symposium on Computer Architecture (ISCA)*, 17-19 de Junho de 1985, Boston, Mass.).

Smith, J. E., G. E. Dermer, B. D. Vanderwarn, S. D. Klinger, C. M. Rozewski, D. L. Fowler, K. R. Scidmore, e J. P. Laudon [1987]. "The ZS-1 central processor", *Proc. Second Int'l. Conf. on Architectural Support for Programming Languages and Oper-ating Systems (ASPLOS)*, 5-8 de Outubro de 1987, Palo Alto, Calif., 199-204.

Smith, M. D., M. Horowitz, e M. S. Lam [1992]. "Efficient superscalar performance through boosting", *Proc. Fifth Int'l. Conf. on Architectural Support for Programming Languages and Operating Systems (ASPLOS)*, 12-15 de Outubro de 1992, Boston, 248-259.

Smith, M. D., M. Johnson, e M. A. Horowitz [1989]. "Limits on multiple instruction issue", *Proc. Third Int'l. Conf. on Architectural Support for Programming Languages and Operating Systems (ASPLOS)*, 3-6 de Abril de 1989, Boston, 290-302.

Smotherman, M. [1989]. "A sequencing-based taxonomy of I/O systems and review of historical machines", *Computer Architecture News* 17:5 (Setembro), 5-15. Reimpresso in *Computer Architecture Readings*, M.D. Hill, N.P. Jouppi, e G. S. Sohi, eds., Morgan Kaufmann, San Francisco, 1999, 451-461.

Sodani, A., e G. Sohi [1997]. "Dynamic instruction reuse", *Proc. 24th Annual Int'l. Symposium on Computer Architecture (ISCA)*, 2-4 de Junho de 1997, Denver, Colo.

Sohi, G. S. [1990]. "Instruction issue logic for high-performance, interruptible, multiple functional unit, pipelined computers", *IEEE Trans. on Computers*, 39:3 (Março), 349-359.

Sohi, G. S., e S. Vajapeyam [1989]. "Tradeoffs in instruction format design for horizon-tal architectures", *Proc. Third Int'l. Conf. on Architectural Support for Programming Languages and Operating Systems (ASPLOS)*, 3-6 de Abril de 1989, Boston, 15-25.

Soundararajan, V., M. Heinrich, B. Verghese, K. Gharachorloo, A. Gupta, e J. L. Hennessy [1998]. "Flexible use of memory for replication/migration in cache-coherent DSM multiprocessors", *Proc. 25th Annual Int'l. Symposium on Computer Architecture (ISCA)*, 3-14 de Julho de 1998, Barcelona, Espanha, 342-355.

SPEC. [1989]. *SPEC Benchmark Suite Release 1.0* (Outubro 2). SPEC. [1994]. *SPEC Newsletter* (Junho).

Sporer, M., F. H. Moss, e C. J. Mathais [1988]. "An introduction to the architecture of the Stellar Graphics supercomputer", *Proc. IEEE COMPCON*, 29 de Fevereiro - 4 de Março de 1988, San Francisco, 464.

Spurgeon, C. [2001]. "Charles Spurgeon's Ethernet Web Site", *www.host.ots.utexas.edu/ethernet/ethernet-home.html*.

Spurgeon, C. [2006]. "Charles Spurgeon's Ethernet Web SITE", *www.ethermanage.com/ethernet/ethernet.html*.

Stenström, P., T. Joe, e A. Gupta [1992]. "Comparative performance evaluation of cache-coherent NUMA and COMA architectures", *Proc. 19th Annual Int'l. Symposium on Computer Architecture (ISCA)*, 19-21 de Maio de 1992, Gold Coast, Australia, 80-91.

Sterling, T. [2001]. *Beowulf PC Cluster Computing with Windows and Beowulf PC Clus-ter Computing with Linux*, MIT Press, Cambridge, Mass.

Stern, N. [1980]. "Who invented the first electronic digital computer?", *Annals of the His-tory of Computing*, 2:4 (Outubro), 375-376.

Stevens, W. R. [1994]. *TCP/IP Illustrated (three volumes)*, Addison-Wesley, Read-ing, Mass.

Stokes, J. [2000]. "Sound and Vision: A Technical Overview of the Emotion Engine", arstechnica.com/reviews/1q00/playstation2/ee-1.html.

Stone, H. [1991]. *High Performance Computers*, Addison-Wesley, New York.

Strauss, W. [1998]. "DSP Strategies 2002", *www.usadata.com/market_research/spr_05/spr_r127-005.htm*.

Strecker, W. D. [1976]. "Cache memories for the PDP-11?", *Proc. Third Annual Int'l. Symposium on Computer Architecture (ISCA)*, 19-21 de Janeiro de 1976, Tampa, Fla., 155-158.

Strecker, W. D. [1978]. "VAX-11/780: A virtual address extension of the PDP-11 fam-ily", *Proc. AFIPS National Computer Conf.*, Junho 5-8, 1978, Anaheim, Calif., 47, 967-980.

Sugumar, R. A., e S. G. Abraham [1993]. "Efficient simulation of caches under optimal replacement with applications to miss characterization", *Proc. ACM SIGMETRICS Conf. on Measurement and Modeling of Computer Systems*, 17-21 de Maio de 1993, Santa Clara, Calif., 24-35.

Sun Microsystems. [1989]. *The SPARC Architectural Manual*, Version 8, Part No. 8001399-09, Sun Microsystems, Santa Clara, Calif.

Sussenguth, E. [1999]. "IBM's ACS-1 Machine", *IEEE Computer*, 22:11 (Novembro).

Swan, R. J., S. H. Fuller, e D. P. Siewiorek [1977]. "Cm*—a modular, multimicroprocessor", *Proc. AFIPS National Computing Conf.*, 13-16 de Junho de 1977, Dallas, Tex., 637-644.

Swan, R. J., A. Bechtolsheim, K. W. Lai, e J. K. Ousterhout [1977]. "The implementation of the Cm* multi-microprocessor", *Proc. AFIPS National Computing Conf.*, 13-16 de Junho de 1977, Dallas, Tex, 645-654.

Swartzlander E. (ed.) [1990]. *Computer Arithmetic*. IEEE Computer Society Press, Los Alamitos, Calif.

Takagi, N., H. Yasuura, e S. Yajima [1985]. "High-speed VLSI multiplication algorithm with a redundant binary addition tree", *IEEE Trans. on Computers* C-34:9, 789-796.

Talagala, N. [2000]. "Characterizing Large Storage Systems: Error Behavior and Performance Benchmarks", Ph.D. dissertation, Computer Science Division, University of California, Berkeley.

Talagala, N., e D. Patterson [1999]. *An Analysis of Error Behavior in a Large Storage System*, Tech. Report UCB//CSD-99-1042, Computer Science Division, University of California, Berkeley.

Talagala, N., R. Arpaci-Dusseau, e D. Patterson [2000]. *Micro-Benchmark Based Extraction of Local and Global Disk Characteristics*, CSD-99-1063, Computer Science Division, University of California, Berkeley.

Talagala, N., S. Asami, D. Patterson, R. Futernick, e D. Hart [2000]. "The art of massive storage: A case study of a Web image archive", *Computer*, Novembro.

Tamir, Y., e G. Frazier [1992]. "Dynamically-allocated multi-queue buffers for VLSI communication switches", *IEEE Trans. on Computers*, 41:6 (Junho), 725-734.

Tanenbaum, A. S. [1978]. "Implications of structured programming for machine architec-ture", *Communications of the ACM*, 21:3 (Março), 237-246.

Tanenbaum, A. S. [1988]. *Computer Networks*, 2nd ed., Prentice Hall, Englewood Cliffs, N.J.

Tang, C. K. [1976]. "Cache design in the tightly coupled multiprocessor system", *Proc. AFIPS National Computer Conf.*, 7-10 de Junho de 1976, New York, 749-753.

Tanqueray, D. [2002]. "The Cray X1 and supercomputer road map", *Proc. 13th Dares-bury Machine Evaluation Workshop*, 11-12 de Dezembro de 2002, Daresbury Laboratories, Daresbury, Cheshire, U.K.

Tarjan, D., S. Thoziyoor, e N. Jouppi [2005]. "HPL Technical Report on CACTI 4.0", www.hpl.hp.com/techeports/2006/HPL=2006+86.html.

Taylor, G. S. [1981]. "Compatible hardware for division and square root", *Proc. 5th IEEE Symposium on Computer Arithmetic*, 18-19 de Maio de 1981, University of Michigan, Ann Arbor, Mich., 127-134.

Taylor, G. S. [1985]. "Radix 16 SRT dividers with overlapped quotient selection stages", *Proc. Seventh IEEE Symposium on Computer Arithmetic*, 4-5 de Junho de 1985, University of Illinois, Urbana, Ill, 64-71.

Taylor, G., P. Hilfinger, J. Larus, D. Patterson, e B. Zorn [1986]. "Evaluation of the SPUR LISP architecture", *Proc. 13th Annual Int'l. Symposium on Computer Architec-ture (ISCA)*, 2 - 5 de Junho de 1986, Tóquio.

Taylor, M. B., W. Lee, S. P. Amarasinghe, e A. Agarwal [2005]. "Scalar operand networks", *IEEE Trans. on Parallel and Distributed Systems*, 16:2 (Fevereiro), 145-162.

Tendler, J. M., J. S. Dodson, J. S. Fields, Jr., H. Le, e B. Sinharoy [2002]. "Power4 system microarchitecture", *IBM J. Research and Development* 46:1, 5-26.

Texas Instruments. [2000]. "History of Innovation: 1980s", www.ti.com/corp/docs/company/history/1980s.shtml.

Tezzaron Semiconductor. [2004]. *Soft Errors in Electronic Memory*, White Paper, Tezzaron Semiconductor, Naperville, Ill. (*http://www.tezzaron.com/about/papers/soft_errors_1_1_secure.pdf*).

Thacker, C. P., E. M. McCreight, B. W. Lampson, R. F. Sproull, e D. R. Boggs [1982]. Alto: A personal computer, in D. P. Siewiorek, C. G. Bell, e A. Newell, eds., *Computer Structures: Principles and Examples*, McGraw-Hill, New York, 549-572.

Thadhani, A. J. [1981]. "Interactive user productivity", *IBM Systems J.* 20:4, 407-423.

Thekkath, R., A. P. Singh, J. P. Singh, S. John, e J. L. Hennessy [1997]. "An evaluation of a commercial CC-NUMA architecture—the CONVEX Exemplar SPP1200", *Proc. 11th Int'l. Parallel Processing Symposium (IPPS)*, 1-7 de Abril de 1997, Genebra, Suíça.

Thorlin, J. F. [1967]. "Code generation for PIE (parallel instruction execution) comput-ers", *Proc. Spring Joint Computer Conf.*, 18-20 de Abril de 1967, Atlantic City, N.J., 27.

Thornton, J. E. [1964]. "Parallel operation in the Control Data 6600", *Proc. AFIPS Fall Joint Computer Conf., Part II*, 27-29 de Outubro 27-29 de 1964, San Francisco, 26, 33-40.

Thornton, J. E. [1970]. *Design of a Computer, the Control Data 6600*, Scott, Foresman, Glenview, Ill.

Tjaden, G. S., e M. J. Flynn [1970]. "Detection and parallel execution of independent instructions", *IEEE Trans. on Computers*, C-19:10 (Outubro), 889-895.

Tomasulo, R. M. [1967]. "An efficient algorithm for exploiting multiple arithmetic units", *IBM J. Research and Development*, 11:1 (Janeiro), 25-33.

Torrellas, J., A. Gupta, e J. Hennessy [1992]. "Characterizing the caching and synchroni-zation performance of a multiprocessor operating system", *Proc. Fifth Int'l. Conf. on Architectural Support for Programming Languages and Operating Systems (ASPLOS)*, 12 - 15 de Outubro de 1992, Boston (*SIGPLAN Notices* 27:9 (Setembro), 162-174).

Touma, W. R. [1993]. *The Dynamics of the Computer Industry: Modeling the Supply of Workstations and Their Components*, Kluwer Academic, Boston.

Tuck, N., e D. Tullsen [2003]. "Initial observations of the simultaneous multithreading Pentium 4 processor", *Proc. 12th Int. Conf. on Parallel Architectures and Compilation Techniques (PACT'03)*, 27 de Setembro - 1 de Outubro de 2003, New Orleans, LA, 26-34.

Tullsen, D. M., S. J. Eggers, e H. M. Levy [1995]. "Simultaneous multithreading: Maximizing on-chip parallelism", *Proc. 22nd Annual Int'l. Symposium on Computer Architecture (ISCA)*, 22-24 de Junho de 1995, Santa Margherita, Itália, 392-403.

Tullsen, D. M., S. J. Eggers, J. S. Emer, H. M. Levy, J. L. Lo, e R. L. Stamm [1996]. "Exploiting choice: Instruction fetch and issue on an implementable simultaneous multithreading processor", *Proc. 23rd Annual Int'l. Symposium on Computer Archi-tecture (ISCA)*, 22-24 de Maio de 1996, Philadelphia, Penn, 191-202.

Ungar, D., R. Blau, P. Foley, D. Samples, e D. Patterson [1984]. "Architecture of SOAR: Smalltalk on a RISC", *Proc. 11th Annual Int'l. Symposium on Computer Architecture (ISCA)*, 5-7 de Junho de 1984, Ann Arbor, Mich, 188-197.

Unger, S. H. [1958]. "A computer oriented towards spatial problems", *Proc. Institute of Radio Engineers*, 46:10 (Outubro), 1744-1750.

Vahdat, A., M. Al-Fares, N. Farrington, R. Niranjan Mysore, G. Porter, e S. Radhakrishnan [2010]. "Scale-Out Networking in the Data Center", *IEEE Micro* 30:4 (July/August), 29-41.

Vaidya, A. S., A. Sivasubramaniam, e C. R. Das [1997]. "Performance benefits of vir-tual channels and adaptive routing: An application-driven study", *Proc. ACM/IEEE Conf. on Supercomputing*, 16-21 de Novembro de 1997, San Jose, Calif.

Vajapeyam, S. [1991]. "Instruction-Level Characterization of the Cray Y-MP Processor", tese de doutorado, Computer Sciences Department, University of Wisconsin-Madison.

van Eijndhoven, J. T. J., F. W. Sijstermans, K. A. Vissers, E. J. D. Pol, M. I. A. Tromp, P. Struik, R. H. J. Bloks, P. van der Wolf, A. D. Pimentel, e H. P. E. Vranken [1999]. "Trimedia CPU64 architecture", *Proc. IEEE Int'l. Conf. on Computer Design: VLSI in Computers and Processors (ICCD'99)*, 10-13 de Outubro de 1999, Austin, Tex., 586-592.

Van Vleck, T. [2005]. "The IBM 360/67 and CP/CMS", http://www.multicians.org/thvv/360-67.html.

von Eicken, T., D. E. Culler, S. C. Goldstein, e K. E. Schauser [1992]. "Active Messages: A mechanism for integrated communication and computation", *Proc. 19th Annual Int'l. Symposium on Computer Architecture (ISCA)*, 19-21 de Maio de 1992, Gold Coast, Australia.

Waingold, E., M. Taylor, D. Srikrishna, V. Sarkar, W. Lee, V. Lee, J. Kim, M. Frank, P. Finch, R. Barua, J. Babb, S. Amarasinghe, e A. Agarwal [1997]. "Baring it all to software: Raw Machines", *IEEE Computer*, 30:(Setembro), 86-93.

Wakerly, J. [1989]. *Microcomputer Architecture and Programming*, Wiley, New York.

Wall, D. W. [1991]. "Limits of instruction-level parallelism", *Proc. Fourth Int'l. Conf. on Architectural Support for Programming Languages and Operating Systems (ASPLOS)*, 8-11 de Abril de 1991, Palo Alto, Calif., 248-259.

Wall, D.W. [1993]. *Limits of Instruction-Level Parallelism*, Research Rep. 93/6, Western Research Laboratory, Digital Equipment Corp., Palo Alto, Calif.

Walrand, J. [1991]. *Communication Networks: A First Course*, Aksen Associates/Irwin, Homewood, Ill.

Wang, W. -H., J. -L. Baer, e H. M. Levy [1989]. "Organization and performance of a two-level virtual-real cache hierarchy", *Proc. 16th Annual Int'l. Symposium on Com-puter Architecture (ISCA)*, 28 de Maio - 1 de Junho de 1989, Jerusalem, 140-148.

Watanabe, T. [1987]. "Architecture and performance of the NEC supercomputer SX sys-tem", *Parallel Computing* 5, 247-255.

Waters, F. (ed.) [1986]. *IBM RT Personal Computer Technology*, SA 23-1057, IBM, Aus-tin, Tex.

Watson, W. J. [1972]. "The TI ASC—a highly modular and flexible super processor archi-tecture", *Proc. AFIPS Fall Joint Computer Conf.*, 5-7 de Dezembro de 1972, Anaheim, Calif., 221-228.

Weaver, D. L., e T. Germond [1994]. *The SPARC Architectural Manual, Version 9*, Prentice Hall, Englewood Cliffs, N.J.

Weicker, R. P. [1984]. "Dhrystone: A synthetic systems programming benchmark", *Com-munications of the ACM*, 27:10 (Outubro), 1013-1030.

Weiss, S., e J. E. Smith [1984]. "Instruction issue logic for pipelined supercomputers", *Proc. 11th Annual Int'l. Symposium on Computer Architecture (ISCA)*, 5-7 de Junho de 1984, Ann Arbor, Mich, 110-118.

Weiss, S., e J. E. Smith [1987]. "A study of scalar compilation techniques for pipelined supercomputers", *Proc. Second Int'l. Conf. on Architectural Support for Program-ming Languages and Operating Systems (ASPLOS)*, 5-8 de Outubro de 1987, Palo Alto, Calif., 105-109.

Weiss, S., e J. E. Smith [1994]. *Power and PowerPC*, Morgan Kaufmann, San Francisco.

Wendel, D., R. Kalla, J. Friedrich, J. Kahle, J. Leenstra, C. Lichtenau, B. Sinharoy, W. Starke, e V. Zyuban [2010]. "The Power7 processor SoC", *Proc. Int'l. Conf. on IC Design and Technology*, 2-4 de Junho de 2010, Grenoble, França, 71-73.

Weste, N., e K. Eshraghian [1993]. *Principles of CMOS VLSI Design: A Systems Per-spective*, 2nd ed., Reading, Mass: Addison-Wesley.

Wiecek, C. [1982]. "A case study of the VAX 11 instruction set usage for compiler execu-tion", *Proc. Symposium on Architectural Support for Programming Languages and Operating Systems (ASPLOS)*, 1-3 de Março de 1982, Palo Alto, Calif, 177-184.

Wilkes, M. [1965]. "Slave memories and dynamic storage allocation", *IEEE Trans. Elec-tronic Computers*, EC-14:2 (Abril), 270-271.

Wilkes, M. V. [1982]. "Hardware support for memory protection: Capability implementa-tions", *Proc. Symposium on Architectural Support for Programming Languages and Operating Systems (ASPLOS)*, 1-3 de Março de 1982, Palo Alto, Calif., 107-116.

Wilkes, M. V. [1985]. *Memoirs of a Computer Pioneer*, MIT Press, Cambridge, Mass.

Wilkes, M. V. [1995]. *Computing Perspectives*, Morgan Kaufmann, San Francisco.

Wilkes, M. V., D. J. Wheeler, e S. Gill [1951]. *The Preparation of Programs for an Electronic Digital Computer*, Addison-Wesley, Cambridge, Mass.

Williams, S., A. Waterman, e D. Patterson [2009]. "Roofline: An insightful visual per-formance model for multicore architectures", *Communications of the ACM*, 52:4 (Abril), 65-76.

Williams, T. E., M. Horowitz, R. L. Alverson, e T. S. Yang [1987]. A self-timed chip for division, in P. Losleben, ed., *1987 Stanford Conference on Advanced Research in VLSI*. MIT Press, Cambridge, Mass.

Wilson, A. W., Jr. [1987]. "Hierarchical cache/bus architecture for shared-memory multi-processors", *Proc. 14th Annual Int'l. Symposium on Computer Architecture (ISCA)*, 2-5 de Junho de 1987, Pittsburgh, Penn, 244-252.

Wilson, R. P., e M. S. Lam [1995]. "Efficient context-sensitive pointer analysis for C programs", *Proc. ACM SIG-PLAN'95 Conf. on Programming Language Design and Implementation*, 18-21 de Junho de 1995, La Jolla, Calif., 1-12.

Wolfe, A., e J. P. Shen [1991]. "A variable instruction stream extension to the VLIW architecture", *Proc. Fourth Int'l. Conf. on Architectural Support for Programming Languages and Operating Systems (ASPLOS)*, 8-11 de Abril de 1991, Palo Alto, Calif., 2-14.

Wood, D. A., e M. D. Hill [1995]. "Cost-effective parallel computing", *IEEE Computer*, 28:2 (Fevereiro), 69-72.

Wulf, W. [1981]. "Compilers and computer architecture", *Computer* 14:7 (Julho), 41-47.

Wulf, W., e C. G. Bell [1972]. "C.mmp—A multi-mini-processor", *Proc. AFIPS Fall Joint Computer Conf.*, 5-7 de Dezembro de 1972, Anaheim, Calif., 765-777.

Wulf, W., e S. P. Harbison [1978]. "Reflections in a pool of processors—an experience report on C.mmp/Hydra", *Proc. AFIPS National Computing Conf.*, 5-8 de Junho de 1978, Anaheim, Calif., 939-951.

Wulf, W. A., e S. A. McKee [1995]. "Hitting the memory wall: Implications of the obvious", *ACM SIGARCH Computer Architecture News*, 23:1 (Março), 20-24.

Wulf, W. A., R. Levin, e S. P. Harbison [1981]. *Hydra/C.mmp: An Experimental Computer System*, McGraw-Hill, New York.

Yamamoto, W., M. J. Serrano, A. R. Talcott, R. C. Wood, e M. Nemirosky [1994]. "Performance estimation of multistreamed, superscalar processors", *Proc. 27th Annual Hawaii Int'l. Conf. on System Sciences*, 4-7 de Janeiro de 1994, Maui, 195-204.

Yang, Y., e G. Mason [1991]. "Nonblocking broadcast switching networks", *IEEE Trans. on Computers*, 40:9 (Setembro), 1005-1015.

Yeager, K. [1996]. "The MIPS R10000 superscalar microprocessor", *IEEE Micro*, 16:2 (Abril), 28-40.

Yeh, T., e Y. N. Patt [1993a]. "Alternative implementations of two-level adaptive branch prediction", *Proc. 19th Annual Int'l. Symposium on Computer Architecture (ISCA)*, 19-21 de Maio de 1992, Gold Coast, Australia, 124-134.

Yeh, T., e Y. N. Patt [1993b]. "A comparison of dynamic branch predictors that use two levels of branch history", *Proc. 20th Annual Int'l. Symposium on Computer Architec-ture (ISCA)*, 16-19 de Maio de 1993, San Diego, Calif., 257-266.

Índice

Referências de página em negrito representam figuras e tabelas.

Números

A

B

C

E

F

G

H

J

M

N

O

P

Q

R

S

U

W

X

Y

Z

Tradução entre os termos de GPU no livro e os termos NVIDIA e OpenCL oficiais.

Tipo	Nome mais descritivo usado neste livro	Termo oficial CUDA/ NVIDIA	Definição do livro e termos OpenCL	Definição oficial CUDA/NVIDIA
Abstrações de programa	**Loop vetorizável**	**Grid**	Um loop vetorizável, executado na GPU, composto de um ou mais blocos de threads (corpos de loop vetorizado) que podem ser executados em paralelo. O nome OpenCL é “faixa de índices”	Um grid é um array de blocos de thread que pode ser executado ao mesmo tempo, sequencialmente, ou uma mistura dos dois.
	Corpo do loop vetorizado	**Bloco de threads**	Um loop vetorizado executado em um “multiprocessador SIMD multithreaded” (processador SIMD multithreaded), composto de um ou mais “warps” (ou threads de instruções SIMD). Estes “warps” (threads SIMD) podem se comunicar através da “Memória compartilhada” (memória local). A OpenCL chama um bloco de threads um “grupo de trabalho”.	Um bloco de threads é um array de threads CUDA que são executados ao mesmo tempo e podem cooperar e se comunicar através da memória compartilhada e sincronização de barreira. Um bloco de threads tem um ID de bloco de threads dentro do seu grid.
	Sequência de operações de pista SIMD	**Thread CUDA**	Um corte vertical de um “warp” (ou thread de instruções SIMD) correspondendo a um elemento executado por um “processador de thread” (ou pista SIMD). O resultado é armazenado dependendo da máscara. A OpenCL chama um thread CUDA de “item de trabalho”.	Um thread CUDA é um thread leve que executa um programa sequencial que pode cooperar com outros threads CIDA sendo executados no mesmo bloco de threads. Um thread CUDA tem um ID de thread dentro do seu bloco de threads.
Objeto de máquina	**Um thread de instruções SIMD**	**Warp**	Um thread tradicional, mas contém somente instruções SIMD que são executados em um “multiprocessador streaming” (processador SIMD multithreaded). Os resultados são armazenados dependendo de uma máscara por elemento.	Um warp é um conjunto de threads CUDA paralelos (p. ex., 32) que executa a mesma instrução ao mesmo tempo em um processador SIMT/SIMD multithreaded.
	Instrução SIMD	**Instrução PTX**	Uma única instrução SIMD executada através dos “processadores de thread” (pistas SIMD).	Uma instrução PTX especifica uma instrução executada por um thread CUDA.
Hardware de processamento	**Processador SIMD multithread**	**Multiprocessador de streaming**	Um processador SIMD multithreaded que executa “warps” (threads de instruções SIMD), independente de outros processadores SIMD. A OpenCL o chama “Unidade computacional”. Entretanto, o programador CUDA escreve programas para uma pista em vez de para um “vetor” de várias pistas SIMD.	Um multiprocessador de streaming (SM) é um processador SIMT/SIMD multithreaded que executa warps de threads CUDA. Um programa SIMT especifica a execução de um thread CUDA em vez de um vetor de diversas pistas SIMD.
	Escalonador de blocos de threads	**Engine Giga thread**	Designa múltiplos “blocos de threads” (corpos de loops vetorizados) a “multiprocessadores de streaming” (processadores SIMD multithreaded).	Distribui e escalona blocos de threads de um grid para multiprocessadores de streaming conforme os recursos ficam disponíveis.
	Escalonador de Threads SIMD	**Escalonador warp**	Unidade de hardware que escalona e despacha “warps” (threads de instruções SIMD) quando elas estão prontas para ser executadas. Inclui um scoreboard para rastrear a execução de “warps” (threads SIMD).	Um escalonador de warp em um multiprocessador de streaming escalona warps para execução quando sua próxima instrução está pronta para ser executada.
	Pista SIMD	**Processador de thread**	Uma pista SIMD que executa as operações em “warp” (thread de instruções SIMD) em um único elemento. Os resultados são armazenados dependendo da máscara. A OpenCL o chama “Elemento de processamento”.	Um processador de thread é um caminho de dados e arquivo registrador de multiprocessador de streaming que executa operações para uma ou mais pistas de um warp.

Tipo	Nome mais descritivo usado neste livro	Termo oficial CUDA/ NVIDIA	Definição do livro e termos OpenCL	Definição oficial CUDA/NVIDIA
Hardware de memória	**Memória da GPU**	**Memória global**	Memória DRAM acessível por todos os "multiprocessadores de streaming" (processadores SIMD multithreaded) em uma GPU. A OpenGL o chama "Memória global".	A memória global é acessível por todos os threads CUDA em qualquer bloco de threads em qualquer grid. Implementado como uma região da DRAM e pode ser colocado no cache.
	Memória privativa	**Memória local**	Parte da memória DRAM privativa de cada "processador de thread" (pista SIMD). A OpenCL o chama "Memória Privativa".	Memória privada "local para o thread" para um thread CUDA. Implementado como uma região da DRAM no cache.
	Memória local	**Memória comparti-lhada**	SDRAM local rápida para um "multipro-cessador de streaming" (processador SIMD multithreaded), indisponível para outros multiprocessadores de strea-ming. A OpenCL o chama "Memória local".	Memória SRAM rápida compartilhada pelos threads CUDA que compõe um bloco de threads, e privado para este bloco de threads. Usado para comunicação entre threads CUDA em um bloco de threads nos pontos de sincronização de barreira.
	Registradores de pista SIMD	**Registrado-res**	Registradores em um único "proces-sador de thread" (pista SIMD), alocados através de todo um "bloco de threads" (corpo de loop vetorizado).	Registradores privados para um thread CUDA. Implementado com um arquivo registrador multithreaded para certas pistas de diversos warps para cada processador de thread.